

Compiler construction

Contents

Articles

Introduction	1
Compiler construction	1
Compiler	2
Interpreter	10
History of compiler writing	13
Lexical analysis	21
Lexical analysis	21
Regular expression	25
Regular expression examples	36
Finite-state machine	40
Preprocessor	50
Syntactic analysis	53
Parsing	53
Lookahead	57
Symbol table	60
Abstract syntax	62
Abstract syntax tree	63
Context-free grammar	64
Terminal and nonterminal symbols	76
Left recursion	78
Backus–Naur Form	82
Extended Backus–Naur Form	85
TBNF	90
Top-down parsing	90
Recursive descent parser	92
Tail recursive parser	97
Parsing expression grammar	99
LL parser	105
LR parser	113
Parsing table	122
Simple LR parser	124

Canonical LR parser	125
GLR parser	127
LALR parser	128
Recursive ascent parser	130
Parser combinator	138
Bottom-up parsing	140
Chomsky normal form	146
CYK algorithm	148
Simple precedence grammar	151
Simple precedence parser	152
Operator-precedence grammar	154
Operator-precedence parser	157
Shunting-yard algorithm	161
Chart parser	171
Earley parser	172
The lexer hack	176
Scannerless parsing	177
Semantic analysis	180
Attribute grammar	180
L-attributed grammar	182
LR-attributed grammar	182
S-attributed grammar	183
ECLR-attributed grammar	183
Intermediate representation	184
Intermediate language	185
Control flow graph	187
Basic block	189
Call graph	191
Data-flow analysis	194
Use-define chain	199
Live variable analysis	202
Reaching definition	204
Three address code	205
Static single assignment form	206
Dominator	212
C3 linearization	214
Intrinsic function	215

Aliasing	216
Alias analysis	218
Array access analysis	220
Pointer analysis	220
Escape analysis	221
Shape analysis	222
Loop dependence analysis	224
Program slicing	227
Code optimization	230
Compiler optimization	230
Peephole optimization	241
Copy propagation	244
Constant folding	245
Sparse conditional constant propagation	247
Common subexpression elimination	248
Partial redundancy elimination	249
Global value numbering	250
Strength reduction	251
Bounds-checking elimination	262
Inline expansion	263
Return value optimization	266
Dead code	269
Dead code elimination	270
Unreachable code	272
Redundant code	274
Jump threading	275
Superoptimization	275
Loop optimization	276
Induction variable	278
Loop fission	281
Loop fusion	282
Loop inversion	283
Loop interchange	285
Loop-invariant code motion	286
Loop nest optimization	287
Manifest expression	291
Polytope model	292

Loop unwinding	294
Loop splitting	301
Loop tiling	302
Loop unswitching	304
Interprocedural optimization	305
Whole program optimization	309
Adaptive optimization	309
Lazy evaluation	310
Partial evaluation	314
Profile-guided optimization	315
Automatic parallelization	316
Loop scheduling	318
Vectorization	318
Superword Level Parallelism	326
Code generation	327
Code generation	327
Name mangling	329
Register allocation	338
Chaitin's algorithm	340
Rematerialization	340
Sethi-Ullman algorithm	341
Data structure alignment	343
Instruction selection	351
Instruction scheduling	352
Software pipelining	354
Trace scheduling	358
Just-in-time compilation	358
Bytecode	362
Dynamic compilation	364
Dynamic recompilation	365
Object file	367
Code segment	368
Data segment	369
.bss	371
Literal pool	372
Overhead code	372
Link time	373

Relocation	373
Library	374
Static build	381
Architecture Neutral Distribution Format	382
Development techniques	384
Bootstrapping	384
Compiler correctness	385
Jensen's Device	387
Man or boy test	388
Cross compiler	390
Source-to-source compiler	396
Tools	398
Compiler-compiler	398
PQCC	400
Compiler Description Language	401
Comparison of regular expression engines	403
Comparison of parser generators	409
Lex	420
flex lexical analyser	423
Quex	430
JLex	433
Ragel	434
yacc	435
Berkeley Yacc	436
ANTLR	437
GNU bison	439
Coco/R	449
GOLD	451
JavaCC	456
JetPAG	457
Lemon Parser Generator	460
ROSE compiler framework	461
SableCC	463
Scannerless Boolean Parser	464
Spirit Parser Framework	465
S/SL programming language	467

SYNTAX	468
Syntax Definition Formalism	469
TREE-META	471
Frameworks supporting the polyhedral model	473
Case studies	478
GNU Compiler Collection	478
Java performance	487
Literature	497
Compilers: Principles, Techniques, and Tools	497
Principles of Compiler Design	499
The Design of an Optimizing Compiler	499
References	
Article Sources and Contributors	500
Image Sources, Licenses and Contributors	504
Article Licenses	
License	505

Introduction

Compiler construction

Compiler construction is an area of computer science that deals with the theory and practice of developing programming languages and their associated compilers.

The theoretical portion is primarily concerned with syntax, grammar and semantics of programming languages. One could say that this gives this particular area of computer science a strong tie with linguistics. Some courses on compiler construction will include a simplified grammar of a spoken language that can be used to form a valid sentence for the purposes of providing students with an analogy to help them understand how grammar works for programming languages.

The practical portion covers actual implementation of compilers for languages. Students will typically end up writing the front end of a compiler for a simplistic teaching language, such as Micro.

Subfields

- Parsing
- Program analysis
- Program transformation
- Compiler or program optimization
- Code generation

Further reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*.
- Michael Wolfe. *High-Performance Compilers for Parallel Computing*. ISBN 978-0805327304

External links

- The ACE CoSy compiler development suite ^[1], an industry leading C/C++ compiler development suite.
- The LLVM Compiler Infrastructure project ^[2], The leading open source compiler tools project.
- Let's Build a Compiler, by Jack Crenshaw ^[3], An interesting tutorial on compiler construction.
- Compiler Construction at the University of New England ^[4]

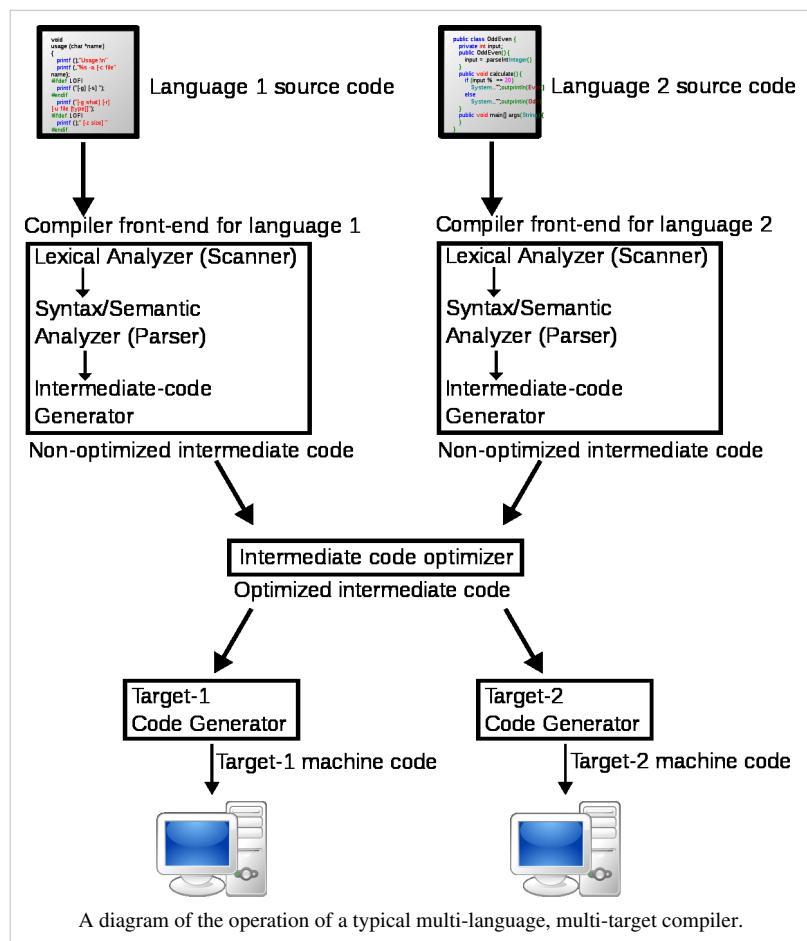
References

- [1] <http://www.ace.nl/compiler/cosy.html>
- [2] <http://llvm.org/>
- [3] <http://compilers.iecc.com/crenshaw/>
- [4] <http://mcs.une.edu.au/~comp319/>

Compiler

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language.



A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software.

The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

History

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler.

Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960.^[1]

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962.^[2] Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Compilers in education

Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum. Such courses are usually supplemented with the implementation of a compiler for an educational programming language. A well-documented example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s.^[3] In spite of its simplicity, the PL/0 compiler introduced several influential concepts to the field:

1. Program development by stepwise refinement (also the title of a 1971 paper by Wirth)^[4]
2. The use of a recursive descent parser
3. The use of EBNF to specify the syntax of a language
4. A code generator producing portable P-code
5. The use of T-diagrams^[5] in the formal description of the bootstrapping problem

Compilation

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different hardware architecture.

With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardwares. Once the target program is generated, the user can execute the program.

The structure of a compiler

Compilers bridge source programs in high-level languages with the underlying hardware. A compiler requires 1) determining the correctness of the syntax of programs, 2) generating correct and efficient object code, 3) run-time organization, and 4) formatting output according to assembler and/or linker conventions. A compiler consists of three main parts: the frontend, the middle-end, and the backend.

The **front end** checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. The frontend then generates an *intermediate representation* or *IR* of the source code for processing by the middle-end.

The **middle end** is where optimization takes place. Typical transformations for optimization are removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. The middle-end generates another IR for the following backend. Most optimization efforts are focused on this part.

The **back end** is responsible for translating the IR from the middle-end into assembly code. The target instruction(s) are chosen for each IR instruction. Register allocation assigns processor registers for the program variables where possible. The backend utilizes the hardware by figuring out how to keep parallel execution units busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

Compiler output

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform*.

A *native* or *hosted* compiler is one which output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

The lower level language that is target of a compiler may itself be a high-level programming language. C, often viewed as some sort of portable assembler, can also be the target language of a compiler. E.g.: Cfront, the original compiler for C++ used C as target language.

Compiled versus interpreted languages

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters.

Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct

arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Hardware compilation

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC). Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively controls the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

Compiler construction

In the early days, the approach taken to compiler design used to be directly affected by the complexity of the processing, the experience of the person(s) designing it, and the resources available.

A compiler for a relatively simple language written by one person might be a single, monolithic piece of software. When the source language is large and complex, and high quality output is required, the design may be split into a number of relatively independent phases. Having separate phases means development can be parceled up into small parts and given to different people. It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

The division of the compilation processes into phases was championed by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University. This project introduced the terms *front end*, *middle end*, and *back end*.

All but the smallest of compilers have more than two phases. However, these phases are usually regarded as being part of the front end or the back end. The point at which these two *ends* meet is open to debate. The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code).

The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

This front-end/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs. Practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

One-pass versus multi-pass compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one-pass compilers generally perform compilations faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's DOALL statements).
- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
 - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
 - Bytecode compilers for Java, Python, and many more are also a subtype of this.
- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)
 - Applications are delivered in bytecode, which is compiled to native machine code just prior to execution.

Front end

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. **Line reconstruction.** Languages which strip their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of ALGOL and Coral 66) are examples of stripped languages which compilers would have a *Line Reconstruction* phase.
2. Lexical analysis breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language,

so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.

3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Back end

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.
3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes (see also Sethi-Ullman algorithm).

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Compiler correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification. Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Related techniques

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an *assembler*, with the inverse program known as a *disassembler*.

A program that translates from a low level language to a higher level one is a *decompiler*.

A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, *language converter*, or *language rewriter*. The last term is usually applied to translations that do not involve a change of language.

International conferences and organizations

Every year, the **European Joint Conferences on Theory and Practice of Software** (ETAPS) sponsors the **International Conference on Compiler Construction** (CC), with papers from both the academic and industrial sectors.^[6]

Notes

- [1] "IP: The World's First COBOL Compilers" (<http://www.interesting-people.org/archives/interesting-people/199706/msg00011.html>). interesting-people.org. 12 June 1997. .
- [2] T. Hart and M. Levin. "The New Compiler, AIM-39 - CSAIL Digital Archive - Artificial Intelligence Laboratory Series" (<ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf>). publications.ai.mit.edu. .
- [3] "The PL/0 compiler/interpreter" (<http://www.246.dk/pl0.html>). .
- [4] "The ACM Digital Library" (<http://www.acm.org/classics/dec95/>). .
- [5] T diagrams were first introduced for describing bootstrapping and cross-compiling compilers in McKeeman et al. *A Compiler Generator* (1971). Conway described the broader concept before that with his UNCOL in 1958, to which Bratman added in 1961: H. Bratman, "An alternate form of the 'UNCOL diagram'", Comm. ACM 4 (March 1961) 3, p. 142. Later on, others, including P.D. Terry, gave an explanation and usage of T-diagrams in their textbooks on the topic of compiler construction. Cf. Terry, 1997, Chapter 3 (<http://scifac.ru.ac.za/compilers/cha03g.htm>). T-diagrams are also now used to describe client-server interconnectivity on the World Wide Web: cf. Patrick Closen, et al. 1997: *T-Diagrams as Visual Language to Illustrate WWW Technology* (<http://pu.rbg.informatik.tu-darmstadt.de/docs/HJH-19990217-etal-T-diagrams.doc>), Darmstadt University of Technology, Darmstadt, Germany
- [6] ETAPS (<http://www.etaps.org/>) - European Joint Conferences on Theory and Practice of Software. Cf. "CC" (Compiler Construction) subsection.

References

- Compiler textbook references (<http://www.informatik.uni-trier.de/~ley/db/books/compiler/index.html>) A collection of references to mainstream Compiler Construction Textbooks
- Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D., *Compilers: Principles, Techniques and Tools* (ISBN 0-201-10088-6) link to publisher (<http://www.aw.com/catalog/academic/product/0,4096,0201100886,00.html>). Also known as "The Dragon Book."
- Allen, Frances E., "A History of Language Processor Technology in IBM" (<http://www.research.ibm.com/journal/rd/255/ibmrd2505Q.pdf>), *IBM Journal of Research and Development*, v.25, no.5, September 1981.

- Allen, Randy; and Kennedy, Ken, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0
- Appel, Andrew Wilson
 - *Modern Compiler Implementation in Java*, 2nd edition. Cambridge University Press, 2002. ISBN 0-521-82060-X
 - *Modern Compiler Implementation in ML* (<http://books.google.com/books?id=8APOYafUt-oC&printsec=frontcover>), Cambridge University Press, 1998. ISBN 0-521-58274-1
- Bornat, Richard, *Understanding and Writing Compilers: A Do It Yourself Guide* (http://www.cs.mdx.ac.uk/staffpages/r_bornat/books/compiling.pdf), Macmillan Publishing, 1979. ISBN 0-333-21732-2
- Cooper, Keith D., and Torczon, Linda, *Engineering a Compiler*, Morgan Kaufmann, 2004, ISBN 1-55860-699-8.
- Leverett; Cattell; Hobbs; Newcomer; Reiner; Schatz; Wulf, *An Overview of the Production Quality Compiler-Compiler Project*, in *Computer* 13(8):38-49 (August 1980)
- McKeeman, William Marshall; Horning, James J.; Wortman, David B., *A Compiler Generator* (<http://www.cs.toronto.edu/XPL/>), Englewood Cliffs, N.J. : Prentice-Hall, 1970. ISBN 0-13-155077-2
- Muchnick, Steven, *Advanced Compiler Design and Implementation* (http://books.google.com/books?id=Pq7pHwG1_OkC&printsec=frontcover&source=gbs_summary_r&cad=0), Morgan Kaufmann Publishers, 1997. ISBN 1-55860-320-4
- Scott, Michael Lee, *Programming Language Pragmatics* (<http://books.google.com/books?id=4LMtA2wOsPcC&printsec=frontcover&dq=Programming+Language+Pragmatics>), Morgan Kaufmann, 2005, 2nd edition, 912 pages. ISBN 0-12-633951-1 (The author's site on this book (<http://www.cs.rochester.edu/~scott/pragmatics/>)).
- Srikant, Y. N.; Shankar, Priti, *The Compiler Design Handbook: Optimizations and Machine Code Generation* (http://books.google.com/books?id=0K_jIsqyNpoC&printsec=frontcover), CRC Press, 2003. ISBN 0-8493-1240-X
- Terry, Patrick D., *Compilers and Compiler Generators: An Introduction with C++* (<http://scifac.ru.ac.za/compilers/conts.htm>), International Thomson Computer Press, 1997. ISBN 1-85032-298-8,
- Wirth, Niklaus, *Construction* (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.3774&rep=rep1&type=pdf>) (ISBN 0-201-40353-6), Addison-Wesley, 1996, 176 pages. Revised November 2005.

External links

- The comp.compilers newsgroup and RSS feed (<http://compilers.iecc.com/>)
- Hardware compilation mailing list (<http://www.jiscmail.ac.uk/lists/hwcomp.html>)
- Practical introduction to compiler construction using flex and yacc (<http://www.onyxbits.de/content/blog/patrick/introduction-compiler-construction-using-flex-and-yacc>)
- Book " Basics of Compiler Design (<http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/>)" by Torben Ægidius Mogensen

Interpreter

In computer science, an **interpreter** normally means a computer program that executes, i.e. *performs*, instructions written in a programming language. An *interpreter* may be a program that either

1. executes the source code directly
2. translates source code into some efficient intermediate representation (code) and immediately executes this
3. explicitly executes stored precompiled code^[1] made by a compiler which is part of the interpreter system

Perl, Python, MATLAB, and Ruby are examples of type 2, while UCSD Pascal is type 3: Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and/or compiler (for JIT systems). Some systems, such as Smalltalk, BASIC, Java and others, may also combine 2 and 3.

While interpreting and compiling are the two main means by which programming languages are implemented, these are not fully distinct categories, one of the reasons being that most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" merely mean that the canonical implementation of that language is an interpreter or a compiler; a high level language is basically an abstraction which is (ideally) independent of particular implementations.

Bytecode interpreters

There is a spectrum of possibilities between interpreting and compiling, depending on the amount of analysis performed before the program is executed. For example, Emacs Lisp is compiled to bytecode, which is a highly compressed and optimized representation of the Lisp source, but is not machine code (and therefore not tied to any particular hardware). This "compiled" code is then interpreted by a bytecode interpreter (itself written in C). The compiled code in this case is machine code for a virtual machine, which is implemented not in hardware, but in the bytecode interpreter. The same approach is used with the Forth code used in Open Firmware systems: the source language is compiled into "F code" (a bytecode), which is then interpreted by a virtual machine.

Control tables - that do not necessarily ever need to pass through a compiling phase - dictate appropriate algorithmic control flow via customized interpreters in similar fashion to bytecode interpreters.

Efficiency

The main disadvantage of interpreters is that when a program is interpreted, it typically runs more slowly than if it had been compiled. The difference in speeds could be tiny or great; often an order of magnitude and sometimes more. It generally takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total time required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyze each statement in the program each time it is executed and then perform the desired action, whereas the compiled code just performs the action within a fixed context determined by the compilation. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

There are various compromises between the development speed when using an interpreter and the execution speed when using a compiler. Some systems (such as some Lisps) allow interpreted and compiled code to call each other and to share variables. This means that once a routine has been tested and debugged under the interpreter it can be compiled and thus benefit from faster execution while other routines are being developed. Many interpreters do not

execute the source code as it stands but convert it into some more compact internal form. Many BASIC interpreters replace keywords with single byte tokens which can be used to find the instruction in a jump table. Others achieve even higher levels of program compaction by using a bit-orientated rather than a byte-orientated program memory structure, where commands tokens may occupy perhaps 5 bits of a 16 bit word, leaving 11 bits for their label or address operands.

An interpreter might well use the same lexical analyzer and parser as the compiler and then interpret the resulting abstract syntax tree.

Advantages and disadvantages of using interpreters

Programmers usually write programs in high level code, which the CPU cannot execute; so this source code has to be converted into machine code. This conversion is done by a compiler or an interpreter. A compiler makes the conversion just once, while an interpreter typically converts it every time a program is executed (or in some languages like early versions of BASIC, every time a single instruction is executed).

Development cycle

During program development, the programmer makes frequent changes to source code. With a compiler, each time the programmer makes a change to the source code, s/he must wait for the compiler to make a compilation of the altered source files, and link all of the binary code files together before the program can be executed. The larger the program, the longer the programmer must wait to see the results of the change. By contrast, a programmer using an interpreter does a lot less waiting, as the interpreter usually just needs to translate the code being worked on to an intermediate representation (or not translate it at all), thus requiring much less time before the changes can be tested.

Distribution

A compiler converts source code into binary instruction for a specific processor's architecture, thus making it less portable. This conversion is made just once, on the developer's environment, and after that the same binary can be distributed to the user's machines where it can be executed without further translation. A cross compiler can generate binary code for the user machine even if it has a different processor than the machine where the code is compiled.

An interpreted program can be distributed as source code. It needs to be translated in each final machine, which takes more time but makes the program distribution independent to the machine's architecture.

Execution environment

An interpreter makes source translations during runtime, meaning that every line of code must be converted to machine --- code each time the program runs. This process slows down the program's execution, something that does not occur for compiled programs. This slowness puts interpreters at a major disadvantage when compared with compilers. Another key disadvantage of an interpreter is that it must be present on the user's machine, (as additional software) to run the program.

Abstract Syntax Tree interpreters

In the spectrum between interpreting and compiling, another approach is transforming the source code into an optimized Abstract Syntax Tree (AST) then executing the program following this tree structure, or using it to generate native code Just-In-Time.^[2] In this approach, each sentence needs to be parsed just once. As an advantage over bytecode, the AST keeps the global program structure and relations between statements (which is lost in a bytecode representation), and when compressed provides a more compact representation.^[3] Thus, using AST has been proposed as a better intermediate format for Just-in-time compilers than bytecode. Also, it allows to perform better analysis during runtime.

However, for interpreters, an AST causes more overhead than a bytecode interpreter, because of nodes related to syntax performing no useful work, of a less sequential representation (requiring traversal of more pointers) and of overhead visiting the tree.^[4]

Just-in-time compilation

Further blurring the distinction between interpreters, byte-code interpreters and compilation is just-in-time compilation (or JIT), a technique in which the intermediate representation is compiled to native machine code at runtime. This confers the efficiency of running native code, at the cost of startup time and increased memory use when the bytecode or AST is first compiled. Adaptive optimization is a complementary technique in which the interpreter profiles the running program and compiles its most frequently-executed parts into native code. Both techniques are a few decades old, appearing in languages such as Smalltalk in the 1980s.^[5]

Just-in-time compilation has gained mainstream attention amongst language implementers in recent years, with Java, the .NET Framework and most modern JavaScript implementations now including JITs.

Punched card interpreter

The term "interpreter" often referred to a piece of unit record equipment that could read punched cards and print the characters in human-readable form on the card. The IBM 550 Numeric Interpreter and IBM 557 Alphabetic Interpreter are typical examples from 1930 and 1954, respectively.

Another definition of interpreter

Instead of producing a target program as a translation, an interpreter performs the operation implied by the source program. For an assignment statement, for example, an interpreter might build a tree and then carry out the operation at the node as it "walks" the tree.

Interpreters are frequently used to execute command languages, since each operator executed in command language is usually an invocation of a complex routine such as an editor or compiler.

Notes and references

[1] In this sense, the CPU is also an interpreter, of machine instructions.

[2] AST intermediate representations (<http://lambda-the-ultimate.org/node/716>), Lambda the Ultimate forum

[3] A Tree-Based Alternative to Java Byte-Codes (<http://citeseer.ist.psu.edu/article/kistler97treebased.html>), Thomas Kistler, Michael Franz

[4] <http://webkit.org/blog/189/announcing-squirrelfish/>

[5] L. Deutsch, A. Schiffman, Efficient implementation of the Smalltalk-80 system (<http://portal.acm.org/citation.cfm?id=800017.800542>), Proceedings of 11th POPL symposium, 1984.

External links

- IBM Card Interpreters (<http://www.columbia.edu/acis/history/interpreter.html>) page at Columbia University
- Theoretical Foundations For Practical 'Totally Functional Programming' (http://www.archive.org/download/TheoreticalFoundationsForPracticaltotallyFunctionalProgramming/33429551_PHD_totalthesis.pdf) (Chapter 7 especially) Doctoral dissertation tackling the problem of formalising what is an interpreter

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

History of compiler writing

In computing, a compiler is a computer program that transforms source code written in a programming language or computer language (the *source language*), into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

Any program written in a high level programming language must be translated to object code before it can be executed, so all programmers using such a language use a compiler or an interpreter. Thus, compilers are very important to programmers. Any improvement to a compiler leads to a large number of improved programs.

Compilers are large and complex programs, but systematic analysis and research by computer scientists has led to a clearer understanding of compiler construction and a large body of theory has been developed around them. Research into compiler construction has led to tools that make it much easier to create compilers, so that today computer science students can create their own small language and develop a simple compiler for it in a few weeks.

First compilers

Software for early computers was primarily written in assembly language. It is usually more productive for a programmer to use a high level language, and programs written in a high level language are able to be reused on different kinds of computers. Even so, it took a while for compilers to become established, because they generated code that did not perform as well as hand-written assembler, they were daunting development projects in their own right, and the very limited memory capacity of early computers created many technical problems for practical compiler implementations.

The first compiler was written by Grace Hopper, in 1952, for the A-0 System language. The term *compiler* was coined by Hopper.^[1] The FORTRAN team led by John W. Backus at IBM is generally credited as having introduced the first complete compiler, in 1957. The first FORTRAN compiler took 18 person-years to create.^[2]

By 1960, an extended Fortran compiler, ALTAC, was also available on the Philco 2000, so it is probable that a Fortran program was compiled for both computer architectures in mid-1960.^[3] The first known demonstrated cross-platform high level language was COBOL. In a demonstration in December 1960, a COBOL program was compiled and executed on both the UNIVAC II and the RCA 501.^[1]

The COBOL compiler for the UNIVAC II was probably the first to be written in a high level language, namely FLOW-MATIC, by a team led by Grace Hopper.

Self-hosting compilers

Like any other software, there are benefits from implementing a compiler in a high-level language. In particular, a compiler can be self-hosted - that is, written in the programming language it compiles. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by a compiler written in a different language, or compiled by running the compiler in an interpreter

ELIAC

The **Navy Electronics Laboratory International ALGOL Compiler** or NELIAC was a dialect and compiler implementation of the ALGOL 58 programming language developed by the Naval Electronics Laboratory in 1958.

ELIAC was the brainchild of Harry Huskey — then Chairman of the ACM and a well known computer scientist, and supported by Maury Halstead, the head of the computational center at NEL. The earliest version was implemented on the prototype USQ-17 computer (called the Countess) at the laboratory. It was the world's first self-compiling compiler. This means that the compiler was first coded in simplified form in assembly language (the *bootstrap*), and

then re-written in its own language, compiled by this bootstrap compiler, and re-compiled by itself, making the bootstrap obsolete.

Lisp

The first self-hosting compiler (excluding assemblers) was written for Lisp by Tim Hart and Mike Levin at MIT in 1962.^[4] They wrote a Lisp compiler in Lisp, testing it inside an existing Lisp interpreter. Once they had improved the compiler to the point where it could compile its own source code, it was self-hosting.^[5]

The compiler as it exists on the standard compiler tape is a machine language program that was obtained by having the S-expression definition of the compiler work on itself through the interpreter. (AI Memo 39)^[5]

This technique is only possible when an interpreter already exists for the very same language that is to be compiled. It borrows directly from the notion of running a program on itself as input, which is also used in various proofs in theoretical computer science, such as the proof that the halting problem is undecidable.

Context-free grammars and parsers

A parser is an important component of a compiler. It parses the source code of a computer programming language to create some form of internal representation. Programming languages tend to be specified in terms of a context-free grammar because fast and efficient parsers can be written for them. Parsers can be written by hand or generated by a parser generator. A context-free grammar provides a simple and precise mechanism for describing the block structure of a program - that is, how programming language constructs are built from smaller blocks. The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky.^[6]

Block structure was introduced into computer programming languages by the ALGOL project (1957–1960), which, as a consequence, also featured a context-free grammar to describe the resulting ALGOL syntax.

Context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar. If a programming language designer is willing to work within some limited subsets of context-free grammars, more efficient parsers are possible.

LR parsing

The LR parser (left to right) was invented by Donald Knuth in 1965 in a paper, "On the Translation of Languages from Left to Right". An **LR parser** is a parser that reads input from **Left** to right (as it would appear if visually displayed) and produces a **Rightmost** derivation. The term **LR(k) parser** is also used, where k refers to the number of unconsumed "look ahead" input symbols that are used in making parsing decisions.

Knuth proved that LR(k) grammars can be parsed with an execution time essentially proportional to the length of the program, and that every LR(k) grammar for $k > 1$ can be mechanically transformed into an LR(1) grammar for the same language. In other words, it is only necessary to have one symbol lookahead to parse any deterministic context-free grammar(DCFG).^[7]

Korenjak (1969) was the first to show parsers for programming languages could be produced using these techniques.^[8] Frank DeRemer devised the more practical SLR and LALR techniques, published in his PhD dissertation at MIT in 1969.^[9] ^[10] This was an important breakthrough, because LR(k) translators, as defined by Donald Knuth, were much too large for implementation on computer systems in the 1960s and 1970s.

In practice, LALR offers a good solution, because LALR(1) grammars are more powerful than SLR(1) and LL(1) grammars. LR(1) grammars are more powerful than LALR(1), however, canonical LR(1) parsers can be extremely large in size and are not considered practical. Minimal LR(1) parsers are small in size and comparable to LALR(1) parsers. The syntax of many programming languages are defined by a grammar that is LALR(1), and for this reason LALR parsers are often used by compilers to perform syntax analysis of source code.

A recursive ascent parser implements an LALR parser using mutually-recursive functions rather than tables. Thus, the parser is *directly encoded* in the host language similar to recursive descent. Direct encoding usually yields a parser which is faster than its table-driven equivalent^[11] for the same reason that compilation is faster than interpretation. It is also (in principle) possible to hand edit a recursive ascent parser, whereas a tabular implementation is nigh unreadable to the average human.

Recursive ascent was first described by Thomas Penello in his article "Very fast LR parsing" ^[12] in 1986. The technique was later expounded upon by G.H. Roberts^[13] in 1988 as well as in an article by Leermakers, Augsteijn, Kruseman Aretz^[14] in 1992 in the journal *Theoretical Computer Science*.

LL parsing

An LL parser parses the input from Left to right, and constructs a Leftmost derivation of the sentence (hence LL, as opposed to LR). The class of grammars which are parsable in this way is known as the *LL grammars*. LL grammars are an even more restricted class of context-free grammars than LR grammars. Nevertheless, they are of great interest to compiler writers, because such a parser is simple and efficient to implement.

LL(k) grammars can be parsed by a recursive descent parser which is usually coded by hand.

The design of ALGOL sparked investigation of recursive descent, since the ALGOL language itself is recursive. The concept of recursive descent parsing was discussed in the January 1961 issue of CACM in separate papers by A.A. Grau and Edgar T. "Ned" Irons.^[15]^[16] Richard Waychoff independently conceived of and used recursive descent in the Burroughs ALGOL compiler in March 1961.^[17]

The idea of LL(1) grammars was introduced by Lewis and Stearns (1968).^[18]^[19]

Recursive descent was popularised by Niklaus Wirth with PL/0, an educational programming language used to teach compiler construction in the 1970s.^[20]

LR parsing can handle a larger range of languages than LL parsing, and is also better at error reporting, i.e. it detects syntactic errors when the input does not conform to the grammar as soon as possible.

Earley parser

In 1970, Jay Earley invented what came to be known as the Earley parser. Earley parsers are appealing because they can parse all context-free languages reasonably efficiently.^[21]

Grammar description languages

John Backus proposed "metalinguistic formulas"^[22] ^[23] to describe the syntax of the new programming language IAL, known today as ALGOL 58 (1959). Backus's work was based on the Post canonical system devised by Emil Post.

Further development of ALGOL led to ALGOL 60; in its report (1963), Peter Naur named Backus's notation **Backus Normal Form** (BNF), and simplified it to minimize the character set used. However, Donald Knuth argued that BNF should rather be read as Backus–Naur Form,^[24] and that has become the commonly accepted usage.

Niklaus Wirth defined Extended Backus–Naur Form (EBNF), a refined version of BNF, in the early 1970s for PL/0. Augmented Backus–Naur Form (ABNF) is another variant. Both EBNF and ABNF are widely used to specify the grammar of programming languages, as the inputs to parser generators, and in other fields such as defining communication protocols.

Parser Generators

A parser generator or *compiler-compiler* is a program that takes a description of the formal grammar of a programming language and produces a parser for that language.

The first compiler-compiler to use that name was written by Tony Brooker in 1960 and was used to create compilers for the Atlas computer at the University of Manchester, including the Atlas Autocode compiler. However it was rather different from modern compiler-compilers, and today would probably be described as being somewhere between a highly customisable generic compiler and an extensible-syntax language. The name 'compiler-compiler' was far more appropriate for Brooker's system than it is for most modern compiler-compilers, which are more accurately described as parser generators. It is almost certain that the "Compiler Compiler" name has entered common use due to Yacc rather than Brooker's work being remembered.

In the early 1960s, Robert McClure at Texas Instruments invented a compiler-compiler called TMG, the name taken from "transmogrification".^[25] ^[26] ^[27] ^[28] In the following years TMG was ported to several UNIVAC and IBM mainframe computers.

The Multics project, a joint venture between MIT and Bell Labs, was one of the first to develop an operating system in a high level language. PL/I was chosen as the language, but an external supplier could not supply a working compiler.^[29] The Multics team developed their own subset dialect of PL/I known as **Early PL/I** (EPL) as their implementation language in 1964. TMG was ported to GE-600 series and used to develop EPL by Douglas McIlroy, Robert Morris, and others.

Not long after Ken Thompson wrote the first version of Unix for the PDP-7 in 1969, Doug McIlroy created the new system's first higher-level language: an implementation of McClure's TMG.^[30] TMG was also the compiler definition tool used by Ken Thompson to write the compiler for the B language on his PDP-7 in 1970. B was the immediate ancestor of C.

An early LALR parser generator was called "TWS", created by Frank DeRemer and Tom Pennello.

XPL

XPL is a dialect of the PL/I programming language, developed in 1967, used for the development of compilers for computer languages. It was designed and implemented by a team with William McKeeman^[31], James J. Horning, and David B. Wortman^[32] at Stanford University and the University of California, Santa Cruz. It was first announced at the 1968 Fall Joint Computer Conference in San Francisco, California.^[33] ^[34]

It is the name of both the programming language and the LALR parser generator system (or TWS: translator writing system) based on the language.

XPL featured a relatively simple bottom-up compiler system dubbed MSP (mixed strategy precedence) by its authors. It was bootstrapped through Burroughs Algol onto the IBM System/360 computer. Subsequent implementations of XPL featured an SLR(1) parser.

yacc

yacc is a parser generator developed by Stephen C. Johnson at AT&T for the Unix operating system.^[35] The name is an acronym for "Yet Another Compiler Compiler." It generates an LALR(1) parser based on a grammar written in a notation similar to Backus-Naur Form.

Johnson worked on yacc in the early 1970s at Bell Labs.^[36] He was familiar with TMG and its influence can be seen in yacc and the design of the C programming language. Because Yacc was the default parser generator on most Unix systems, it was widely distributed and used. Developments like GNU Bison are still in use.

The parser generated by yacc requires a lexical analyzer. Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and Yacc.

Cross compilation

A cross compiler runs in one environment but produces object code for another. Cross compilers are used for embedded development, where the target computer has limited capabilities.

An early example of cross compilation was AIMICO, where a FLOW-MATIC program on a UNIVAC II was used to generate assembly language for the IBM 705, which was then assembled on the IBM computer.^[1]

The ALGOL 68C compiler generated *ZCODE* output, that could then be either compiled into the local machine code by a *ZCODE* translator or run interpreted. *ZCODE* is a register-based intermediate language. This ability to interpret or compile *ZCODE* encouraged the porting of ALGOL 68C to numerous different computer platforms.

Optimizing compilers

Compiler optimization is the process of improving the quality of object code without changing the results it produces.

The developers of the first FORTRAN compiler aimed to generate code that was *better* than the average hand-coded assembler, so that customers would actually use their product. In one of the first real compilers, they often succeeded.^[37]

Later compilers, like IBM's Fortran IV compiler, placed more priority on good diagnostics and executing more quickly, at the expense of object code optimization. It wasn't until the IBM 360 series that IBM provided two separate compilers became available: a fast executing code checker, and a slower optimizing one.

Frances E. Allen, working alone and jointly with John Cocke, introduced many of the concepts for optimization. Allen's 1966 paper, *Program Optimization*,^[38] introduced the use of graph data structures to encode program content for optimization^[39]. Her 1970 papers, *Control Flow Analysis*^[40] and *A Basis for Program Optimization*^[41] established *intervals* as the context for efficient and effective data flow analysis and optimization. Her 1971 paper with Cocke, *A Catalog of Optimizing Transformations*,^[42] provided the first description and systematization of optimizing transformations. Her 1973 and 1974 papers on interprocedural data flow analysis extended the analysis to whole programs.^{[43] [44]} Her 1976 paper with Cocke describes one of the two main analysis strategies used in optimizing compilers today.^[45]

Allen developed and implemented her methods as part of compilers for the IBM 7030 Stretch-Harvest and the experimental Advanced Computing System. This work established the feasibility and structure of modern machine- and language-independent optimizers. She went on to establish and lead the PTRAN project on the automatic parallel execution of FORTRAN programs.^[46] Her PTRAN team developed new parallelism detection schemes and created the concept of the program dependence graph, the primary structuring method used by most parallelizing compilers.

Programming Languages and their Compilers by John Cocke and Jacob T. Schwartz, published early in 1970, devoted more than 200 pages to optimization algorithms. It included many of the now familiar techniques such as redundant code elimination and strength reduction.^[47]

Peephole Optimization

Peephole optimization is a very simple but effective optimization technique. It was invented by William McKeeman and published in 1965 in CACM.^[48] It was used in the XPL compiler that McKeeman helped develop.

Capex COBOL Optimizer

Capex Corporation developed the "COBOL Optimizer" in the mid 1970's for COBOL. This type of optimizer depended, in this case, upon knowledge of 'weaknesses' in the standard IBM COBOL compiler, and actually replaced (or patched) sections of the object code with more efficient code. The replacement code might replace a linear table lookup with a binary search for example or sometimes simply replace a relatively 'slow' instruction with a known faster one that was otherwise functionally equivalent within its context. This technique is now known as "Strength reduction". For example on the IBM/360 hardware the **CLI** instruction was, depending on the particular model, between twice and 5 times as fast as a **CLC** instruction for single byte comparisons.^{[49] [50]}

Modern compilers typically provide optimization options, so programmers can choose whether or not to execute an optimization pass.

Diagnostics

When a compiler is given a syntactically incorrect program, a good, clear error message is helpful. From the perspective of the compiler writer, it is often difficult to achieve.

The WATFIV Fortran compiler was developed at the University of Waterloo, Canada in the late 1960s. It was designed to give better error messages than IBM's Fortran compilers of the time. In addition, WATFIV was far more usable, because it combined compiling, linking and execution into one step, whereas IBM's compilers had three separate components to run.

PL/C

PL/C was a computer programming language developed at Cornell University in the early 1970s. While PL/C was a subset of IBM's PL/I language, it was designed with the specific goal of being used for teaching programming. The two researchers and academic teachers who designed PL/C were Richard W. Conway and Thomas R. Wilcox.^[1] They submitted the famous article "Design and implementation of a diagnostic compiler for PL/I" published in the Communications of ACM in March 1973.^[51]

PL/C eliminated some of the more complex features of PL/I, and added extensive debugging and error recovery facilities. The PL/C compiler had the unusual capability of never failing to compile any program, through the use of extensive automatic correction of many syntax errors and by converting any remaining syntax errors to output statements.

References

- [1] (http://www.computerhistory.org/events/lectures/cobol_06121997/index.shtml) The World's First COBOL Compilers
- [2] Backus et al. "The FORTRAN automatic coding system", Proc. AFIPS 1957 Western Joint Computer Conf., Spartan Books, Baltimore 188-198
- [3] (<http://portal.acm.org/citation.cfm?id=808498>) Rosen, Saul. *ALTAC, FORTRAN, and compatibility*. Proceedings of the 1961 16th ACM national meeting
- [4] T. Hart and M. Levin "The New Compiler", AIM-39 (<ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf>) CSAIL Digital Archive - Artificial Intelligence Laboratory Series
- [5] Tim Hart and Mike Levin. "AI Memo 39-The new compiler" (<ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf>). . Retrieved 2008-05-23.
- [6] Chomsky, Noam (Sep 1956). "Three models for the description of language" (<http://ieeexplore.ieee.org/iel5/18/22738/01056813.pdf?isnumber=22738&prod=STD&arnumber=1056813&arnumber=1056813&arSt=+113&ared=+124&arAuthor=+Chomsky,+N.>). *Information Theory, IEEE Transactions* 2 (3): 113–124. doi:10.1109/TIT.1956.1056813. . Retrieved 2007-06-18.
- [7] Knuth, Donald. "On the Translation of Languages from Left to Right" (<http://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf>). . Retrieved 29 May 2011.
- [8] Korenjak, A. "A Practical Method for Constructing LR(k) Processors," Communications of the ACM, Vol. 12, No. 11, 1969
- [9] DeRemer, F. Practical Translators for LR(k) Languages. Ph.D. dissertation, MIT, 1969.
- [10] DeRemer, F. "Simple LR(k) Grammars," Communications of the ACM, Vol. 14, No. 7, 1971.
- [11] Thomas J Penello (1986). "Very fast LR parsing" (<http://portal.acm.org/citation.cfm?id=13310.13326>). .
- [12] <http://portal.acm.org/citation.cfm?id=13310.13326>
- [13] G.H. Roberts (1988). "Recursive ascent: an LR analog to recursive descent" (<http://portal.acm.org/citation.cfm?id=47907.47909>). .
- [14] Leermakers, Augusteijn, Kruseman Aretz (1992). "A functional LR parser" (<http://portal.acm.org/citation.cfm?id=146986.146994>). .
- [15] A.A. Grau, "Recursive processes and ALGOL translation", Commun. ACM, 4, #1, pp. 10-15. Jan. 1961
- [16] Edgar T. Irons, "A syntax-directed compiler for ALGOL 60", Commun. ACM, 4, #1, Jan. 1961, pp. 51-55.
- [17] <http://web.me.com/ianjoyner/Files/Waychoff.pdf>
- [18] P. M. Lewis, R. E. Stearns, "Syntax directed transduction," focs, pp.21-35, 7th Annual Symposium on Switching and Automata Theory (SWAT 1966), 1966
- [19] Lewis, P. and Stearns, R. "Syntax-Directed Transduction," Journal of the ACM, Vol. 15, No. 3, 1968.
- [20] "The PL/0 compiler/interpreter" (<http://www.246.dk/pl0.html>). .
- [21] J. Earley, "An efficient context-free parsing algorithm" (<http://portal.acm.org/citation.cfm?doid=362007.362035>), *Communications of the Association for Computing Machinery*, 13:2:94-102, 1970.
- [22] Backus, J.W. (1959). "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference" (http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf/view). *Proceedings of the International Conference on Information Processing*. UNESCO. pp. 125–132. .
- [23] Farrell, James A. (August 1995). "Extended Backus Naur Form" (<http://www.cs.man.ac.uk/~pjf/farrell/comp2.html#EBNF>). . Retrieved May 11, 2011.
- [24] Donald E. Knuth, "Backus Normal Form vs. Backus Naur Form", Commun. ACM, 7(12):735–736, 1964.
- [25] <http://www.reocities.com/ResearchTriangle/2363/tmg011.html>
- [26] <http://hopl.murdoch.edu.au/showlanguage.prx?exp=242>
- [27] <http://portal.acm.org/citation.cfm?id=806050&dl=ACM&coll=DL&CFID=29658196&CFTOKEN=62044584>
- [28] R. M. McClure, *TMG—A Syntax Directed Compiler* Proc. 20th ACM National Conf. (1965), pp. 262-274.
- [29] <http://multicians.org/pl1.html>
- [30] (<http://cm.bell-labs.com/who/dmr/chist.html>) Dennis M. Ritchie. *The Development of the C Language*
- [31] <http://www.cs.dartmouth.edu/~mckeeman/>
- [32] <http://www.cs.toronto.edu/~dw/>
- [33] McKeeman, William Marshall; Horning, James J.; and Wortman, David B., *A Compiler Generator* (1971), ISBN 978-0131550773.
- [34] Computer Science Department, University of Toronto, "The XPL Programming Language" (<http://www.cs.toronto.edu/XPL/>)
- [35] Johnson, S.C., "Yacc - Yet Another Compiler Compiler", Computing Science Technical Report 32, AT&T Bell Labs, 1975
- [36] http://www.techworld.com.au/article/252319/a-z_programming_languages_yacc/
- [37] <http://compilers.iecc.com/comparch/article/97-10-017>
- [38] F.E. Allen. Program optimization. In Mark I. Halpern and Christopher J. Shaw, editors, *Annual Review in Automatic Programming*, volume 5, pages 239-307. Pergamon Press, New York, 1969.
- [39] Frances E. Allen and John Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1972.
- [40] Frances E. Allen. Control flow analysis. ACM SIGPLAN Notices, 5(7):1-19, July 1970.
- [41] Frances E. Allen. A basis for program optimization. In Proc. IFIP Congress 71, pages 385-390. North-Holland, 1972.
- [42] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1-30. Prentice-Hall, 1971.

- [43] Frances E. Allen. Interprocedural data flow analysis. In Proc. IFIP Congress 74, pages 398-402. North-Holland, 1974.
- [44] Frances E. Allen. A method for determining program data relationships. In Andrei Ershov and Valery A. Nepomniashch, editors, Proc. International Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972, volume 5 of Lecture Notes in Computer Science, pages 299-308. Springer-Verlag, 1974.
- [45] Frances E. Allen and John Cocke. A program data flow analysis procedure. Communications of the ACM, 19(3):137-147, March 1976.
- [46] Vivek Sarkar. The PTRAN Parallel Programming System. In Parallel Functional Programming Languages and Compilers, edited by B. Szymanski, ACM Press Frontier Series, pages 309-391, 1991.
- [47] John Cocke and Jacob T. Schwartz, Programming Languages and their Compilers. Courant Institute of Mathematical Sciences, New York University, April 1970.
- [48] MCKEEMAN, W.M. Peephole optimization. Commun. ACM 8, 7 (July 1965), 443-444
- [49] http://www.bitsavers.org/pdf/ibm/360/A22_6825-1_360instrTiming.pdf
- [50] <http://portal.acm.org/citation.cfm?id=358732&dl=GUIDE&dl=ACM>
- [51] CACM March 1973 pp 169-179.

Further reading

- Backus, John, et al., "The FORTRAN Automatic Coding System" (<http://archive.computerhistory.org/resources/text/Fortran/102663113.05.01.acc.pdf>), Proceedings of the Western Joint Computer Conference, Los Angeles, California, February 1957. Describes the design and implementation of the first FORTRAN compiler by the IBM team.
- Bauer, Friedrich L.; Eickel, Jürgen (Eds.), *Compiler Construction, An Advanced Course*, 2nd ed. Lecture Notes in Computer Science 21, Springer 1976, ISBN 3-540-07542-9
- Cheatham, T. E., and Sattley, K., *Syntax directed compilation*, SJCC p. 31. (1964).
- Cocke, John; Schwartz, Jacob T., *Programming Languages and their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences technical report, New York University, 1969.
- Conway, Melvin E., *Design of a separable transition-diagram compiler*, Communications of the ACM, Volume 6, Issue 7 (July 1963)
- Floyd, R. W., *Syntactic analysis and operator precedence*, Journal of the ACM, Vol. 10, p. 316. (July 1963).
- Gries, David, *Compiler Construction for Digital Computers*, New York : Wiley, 1971. ISBN 047132776X
- Irons, Edgar T., *A syntax directed compiler for ALGOL 60*, Communications of the ACM, Vol. 4, p. 51. (Jan. 1961)
- Knuth, D. E., *On the translation of languages from left to right.*, Information and Control, Vol. 8, p. 607. (1965).
- Knuth, D. E., *RUNCIBLE-algebraic translation on a limited computer*, Communications of the ACM, Vol. 2, p. 18, (Nov. 1959).
- Randell, Brian; Russell, Lawford John, *ALGOL 60 Implementation: The Translation and Use of ALGOL 60 Programs on a Computer*, Academic Press, 1964
- Dick Grune's Annotated Literature Lists - Compiler Construction before 1980 (<http://www.dickgrune.com/Summaries/CS/CompilerConstruction-1979.html>)

Lexical analysis

Lexical analysis

In computer science, **lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer, lexer or scanner**. A lexer often exists as a single function which is called by a parser or another function.

Lexical grammar

The specification of a programming language will often include a set of rules which defines the lexer. These rules usually consist of regular expressions and they define the set of possible character sequences that are used to form individual tokens or lexemes.

In programming languages delimiting blocks with tokens (e.g., "{" and "}") as opposed to off-side rule languages delimiting blocks with indentation, white space is also defined by a regular expression and influences the recognition of other tokens but does not itself contribute tokens. White space is said to be *non-significant* in such languages.

Token

A **token** is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each '(' is matched with a ')'.

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table:

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

Scanner

The first stage, the **scanner**, is usually based on a finite-state machine (FSM). It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as lexemes). For instance, an *integer* token may contain any sequence of numerical digit characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the maximal munch rule, or longest match rule). In some languages the lexeme creation rules are more complicated and may involve backtracking over previously read characters.

Tokenizer

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

Take, for example,

```
The quick brown fox jumps over the lazy dog
```

The string isn't implicitly segmented on spaces, as an English speaker would do. The raw input, the 43 characters, must be explicitly split into the 9 tokens with a given space delimiter (i.e. matching the string " " or regular expression /\s{1} /).

The tokens could be represented in XML,

```
<sentence>
  <word>The</word>
  <word>quick</word>
  <word>brown</word>
  <word>fox</word>
  <word>jumps</word>
  <word>over</word>
  <word>the</word>
  <word>lazy</word>
  <word>dog</word>
</sentence>
```

Or an s-expression,

```
(sentence ((word The) (word quick) (word brown) (word fox) (word jumps)
          (word over) (word the) (word lazy) (word dog)))
```

A lexeme, however, is only a string of characters known to be of a certain kind (e.g., a string literal, a sequence of letters). In order to construct a token, the lexical analyzer needs a second stage, the **evaluator**, which goes over the characters of the lexeme to produce a *value*. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. (Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing. The evaluators for integers, identifiers, and strings can be considerably more complex. Sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments.)

For example, in the source code of a computer program the string

```
net_worth_future = (assets - liabilities);
```

might be converted (with whitespace suppressed) into the lexical token stream:

```
NAME "net_worth_future"
EQUALS
OPEN_PARENTHESIS
NAME "assets"
MINUS
NAME "liabilities"
CLOSE_PARENTHESIS
SEMICOLON
```

Though it is possible and sometimes necessary, due to licensing restrictions of existing parsers or if the list of tokens is small, to write a lexer by hand, lexers are often generated by automated tools. These tools generally accept regular expressions that describe the tokens allowed in the input stream. Each regular expression is associated with a production in the lexical grammar of the programming language that evaluates the lexemes matching the regular expression. These tools may generate source code that can be compiled and executed or construct a state table for a finite-state machine (which is plugged into template code for compilation and execution).

Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, for an English-based language, a NAME token might be any English alphabetical character or an underscore, followed by any number of instances of any ASCII alphanumeric character or an underscore. This could be represented compactly by the string `[a-zA-Z_][a-zA-Z_0-9]*`. This means "any character a-z, A-Z or _, followed by 0 or more of a-z, A-Z, _ or 0-9".

Regular expressions and the finite-state machines they generate are not powerful enough to handle recursive patterns, such as " n opening parentheses, followed by a statement, followed by n closing parentheses." They are not capable of keeping count, and verifying that n is the same on both sides — unless you have a finite set of permissible values for n . It takes a full-fledged parser to recognize such patterns in their full generality. A parser can push parentheses on a stack and then try to pop them off and see if the stack is empty at the end. (see example^[1] in the SICP book).

The Lex programming tool and its compiler is designed to generate code for fast lexical analysers based on a formal description of the lexical syntax. It is not generally considered sufficient for applications with a complicated set of lexical rules and severe performance requirements; for instance, the GNU Compiler Collection (gcc) uses hand-written lexers.

Lexer generator

Lexical analysis can often be performed in a single pass if reading is done a character at a time. Single-pass lexers can be generated by tools such as the classic flex.

The lex/flex family of generators uses a table-driven approach which is much less efficient than the directly coded approach. With the latter approach the generator produces an engine that directly jumps to follow-up states via goto statements. Tools like re2c and Quex have proven (e.g. RE2C - A More Versatile Scanner Generator (1994)^[2]) to produce engines that are between two to three times faster than flex produced engines. It is in general difficult to hand-write analyzers that perform better than engines generated by these latter tools.

The simple utility of using a scanner generator should not be discounted, especially in the developmental phase, when a language specification might change daily. The ability to express lexical constructs as regular expressions facilitates the description of a lexical analyzer. Some tools offer the specification of pre- and post-conditions which are hard to program by hand. In that case, using a scanner generator may save a lot of development time.

Lexical analyzer generators

- ANTLR - ANTLR generates predicated-LL(k) lexers.
- Flex - Alternative variant of the classic 'lex' (C/C++).
- JFlex - a rewrite of JLex.
- Ragel - A state machine and lexical scanner generator with output support for C, C++, Objective-C, D, Java and Ruby source code.

The following lexical analysers can handle Unicode:

- JavaCC - JavaCC generates lexical analyzers written in Java.
- JLex - A Lexical Analyzer Generator for Java.
- Quex - (or 'Quex') A Fast Universal Lexical Analyzer Generator for C and C++.

References

- [1] mitpress.mit.edu (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-31.html#%_sec_5.1.4)
- [2] citeseerx.ist.psu.edu (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.1624>)
- CS 164: Programming Languages and Compilers (Class Notes #2: Lexical) (http://www.cs.berkeley.edu/~hilfingr/cs164/public_html/lectures/note2.pdf)
- *Compiling with C# and Java*, Pat Terry, 2005, ISBN 0-321-26360-X 624
- *Algorithms + Data Structures = Programs*, Niklaus Wirth, 1975, ISBN 0-13-022418-9
- *Compiler Construction*, Niklaus Wirth, 1996, ISBN 0-201-40353-6
- Sebesta, R. W. (2006). Concepts of programming languages (Seventh edition) pp. 177. Boston: Pearson/Addison-Wesley.
- Word Mention Segmentation Task (http://www.gabormelli.com/RKB/Word_Mention_Segmentation_Task) analysis page
- On the applicability of the longest-match rule in lexical analysis (<http://www.cis.nctu.edu.tw/~wuuyang/papers/applicability2002.ps>)

Regular expression

In computing, a **regular expression**, also referred to as **regex** or **regexp**, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- The sequence of characters "car" appearing consecutively in any context, such as in "car", "cartoon", or "bicarbonate"
- The sequence of characters "car" occurring in that order with other characters between them, such as in "Icelander" or "chandler"
- The word "car" when it appears as an isolated word
- The word "car" when preceded by the word "blue" or "red"
- The word "car" when *not* preceded by the word "motor"
- A dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits (for example, "\$100" or "\$245.99").

Regular expressions can be much more complex than these examples.

Utilities provided by Unix distributions—including the editor ed and the filter grep—were the first to popularize the concept of regular expressions. Regular expressions are used by many text editors, utilities, and programming languages to search and manipulate text based on patterns. Some of these languages, including Perl, Ruby, Awk, and Tcl, have fully integrated regular expressions into the syntax of the core language itself. Others like .NET languages, Java, and Python instead provide regular expressions through standard libraries. For other languages, such as Object Pascal, C and C++, non-core libraries are available (However, the upcoming version C++11 introduces regular expressions as part of its Standard Libraries).

As an example of the syntax, the regular expression `\bex` can be used to search for all instances of the string "ex" that occur after "word boundaries" (signified by the `\b`). Thus `\bex` will find the matching string "ex" in two possible locations, (1) at the beginning of words, and (2) between two characters in a string, where the first is not a word character and the second is a word character. For instance, in the string "Texts for experts", `\bex` matches the "ex" in "experts" but not in "Texts" (because the "ex" occurs inside a word and not immediately after a word boundary).

Many modern computing systems provide wildcard characters in matching filenames from a file system. This is a core capability of many command-line shells and is also known as globbing. Wildcards differ from regular expressions in generally expressing only limited forms of patterns.

Basic concepts

A regular expression, often called a pattern, is an expression that describes a set of strings. They are usually used to give a concise description of a set, without having to list all elements. For example, the set containing the three strings "Handel", "Händel", and "Haendel" can be described by the pattern `H (ä | ae?) ndel` (or alternatively, it is said that the pattern *matches* each of the three strings). In most formalisms, if there is any regex that matches a particular set then there is an infinite number of such expressions. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A vertical bar separates alternatives. For example, `gray | grey` can match "gray" or "grey".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "gray" and "grey".

Quantification

A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark ?, the asterisk * (derived from the Kleene star), and the plus sign + (Kleene cross).

- ? The question mark indicates there is *zero or one* of the preceding element. For example, `color?r` matches both "color" and "colour".
- * The asterisk indicates there is *zero or more* of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbcc", and so on.
- + The plus sign indicates there is *one or more* of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbcc", and so on, but not "ac".

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, -, ×, and ÷. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

History

The origins of regular expressions lie in automata theory and formal language theory, both of which are part of theoretical computer science. These fields study models of computation (automata) and ways to describe and classify formal languages. In the 1950s, mathematician Stephen Cole Kleene described these models using his mathematical notation called *regular sets*.^[1] The SNOBOL language was an early implementation of pattern matching, but not identical to regular expressions. Ken Thompson built Kleene's notation into the editor QED as a means to match patterns in text files. He later added this capability to the Unix editor ed, which eventually led to the popular search tool grep's use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: `g/re/p` where *re* stands for regular expression^[2]). Since that time, many variations of Thompson's original adaptation of regular expressions have been widely used in Unix and Unix-like utilities including expr, AWK, Emacs, vi, and lex.

Perl and Tcl regular expressions were derived from a regex library written by Henry Spencer, though Perl later expanded on Spencer's library to add many new features.^[3] Philip Hazel developed PCRE (Perl Compatible Regular Expressions), which attempts to closely mimic Perl's regular expression functionality and is used by many modern tools including PHP and Apache HTTP Server. Part of the effort in the design of Perl 6 is to improve Perl's regular expression integration, and to increase their scope and capabilities to allow the definition of parsing expression grammars.^[4] The result is a mini-language called Perl 6 rules, which are used to define Perl 6 grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regular expressions, but also allow BNF-style definition of a recursive descent parser via sub-rules.

The use of regular expressions in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s when industry standards like ISO SGML (precursored by ANSI "GCA 101-1983") consolidated. The kernel of the structure specification language standards are regular expressions. Simple use is evident in the DTD element group syntax.

See also Pattern matching: History.

Formal language theory

Definition

Regular expressions describe regular languages in formal language theory. They have thus the same expressive power as regular grammars. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory.^[5]^[6] Given a finite alphabet Σ , the following constants are defined:

- (*empty set*) \emptyset denoting the set \emptyset .
- (*empty string*) ϵ denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) a in Σ denoting the set containing only the character a .

The following operations are defined:

- (*concatenation*) RS denoting the set $\{ \alpha\beta \mid \alpha \text{ in } R \text{ and } \beta \text{ in } S \}$. For example $\{ "ab", "c" \} \{ "d", "ef" \} = \{ "abd", "abef", "cd", "cef" \}$.
- (*alternation*) $R \mid S$ denoting the set union of R and S . For example $\{ "ab", "c" \} \{ "ab", "d", "ef" \} = \{ "ab", "c", "d", "ef" \}$.
- (*Kleene star*) R^* denoting the smallest superset of R that contains ϵ and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from R . For example, $\{ "0", "1" \}^*$ is the set of all finite binary strings (including the empty string), and $\{ "ab", "c" \}^* = \{ \epsilon, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... \}$.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then parentheses may be omitted. For example, $(ab)c$ can be written as abc , and $a \mid (b(c^*))$ can be written as $a \mid b c^*$. Many textbooks use the symbols \cup , $+$, or \vee for alternation instead of the vertical bar.

Examples:

- $a \mid b^*$ denotes $\{ \epsilon, a, b, bb, bbb, ... \}$
- $(a \mid b)^*$ denotes the set of all strings with no symbols other than a and b , including the empty string: $\{ \epsilon, a, b, aa, ab, ba, bb, aaa, ... \}$
- $ab^*(c \mid \epsilon)$ denotes the set of strings starting with a , then zero or more bs and finally optionally a c : $\{ a, ac, ab, abc, abb, abbc, ... \}$

Expressive power and compactness

The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers $?$ and $+$, which can be expressed as follows: $a^+ = aa^*$, and $a^? = (a \mid \epsilon)$. Sometimes the complement operator is added, to give a *generalized regular expression*; here R^c matches all strings over Σ^* that do not match R . In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.^[7]^[8]

Regular expressions in this sense can express the regular languages, exactly the class of languages accepted by deterministic finite automata. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows exponentially in the size of the shortest equivalent regular expressions. The standard example are here the languages L_k consisting of all strings over the alphabet $\{a,b\}$ whose k^{th} -last letter equals a . On the one hand, a regular expression describing L_4 is given by $(a|b)^*a(a|b)(a|b)(a|b)$. Generalizing this pattern to L_k gives the expression

$$(a|b)^*a\underbrace{(a|b)(a|b)\cdots(a|b)}_{k-1 \text{ times}}.$$

On the other hand, it is known that every deterministic finite automaton accepting the language L_k must have at least 2^k states. Luckily, there is a simple mapping from regular expressions to the more general nondeterministic finite automata (NFAs) that does not lead to such a blowup in size; for this reason NFAs are often used as alternative representations of regular languages. NFAs are a simple variation of the type-3 grammars of the Chomsky hierarchy.^[5]

Finally, it is worth noting that many real-world "regular expression" engines implement features that cannot be described by the regular expressions in the sense of formal language theory; see below for more on this.

Deciding equivalence of regular expressions

As seen in many of the examples above, there is more than one way to construct a regular expression to achieve the same results.

It is possible to write an algorithm which for two given regular expressions decides whether the described languages are essentially equal, reduces each expression to a minimal deterministic finite state machine, and determines whether they are isomorphic (equivalent).

The redundancy can be eliminated by using Kleene star and set union to find an interesting subset of regular expressions that is still fully expressive, but perhaps their use can be restricted. This is a surprisingly difficult problem. As simple as the regular expressions are, there is no method to systematically rewrite them to some normal form. The lack of axiom in the past led to the star height problem. Recently, Dexter Kozen axiomatized regular expressions with Kleene algebra.^[9]

Syntax

A number of special characters or meta characters are used to denote actions or delimit groups; but it is possible to force these special characters to be interpreted as normal characters by preceding them with a defined escape character, usually the backslash "\". For example, a dot is normally used as a "wild card" metacharacter to denote any character, but if preceded by a backslash it represents the dot character itself. The pattern `c.t` matches "cat", "cot", "cut", and non-words such as "czt" and "c.t"; but `c\.t` matches only "c.t". The backslash also escapes itself, i.e., two backslashes are interpreted as a literal backslash character.

POSIX

POSIX Basic Regular Expressions

Traditional Unix regular expression syntax followed common conventions but often differed from tool to tool. The IEEE POSIX Basic Regular Expressions (BRE) standard (released alongside an alternative flavor called Extended Regular Expressions or ERE) was designed mostly for backward compatibility with the traditional (Simple Regular Expression) syntax but provided a common standard which has since been adopted as the default syntax of many Unix regular expression tools, though there is often some variation or additional features. Many such tools also provide support for ERE syntax with command line arguments.

In the BRE syntax, most characters are treated as literals — they match only themselves (e.g., `a` matches "a"). The exceptions, listed below, are called metacharacters or metasequences.

Metacharacter	Description
.	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor, character encoding, and platform specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, <code>a . c</code> matches "abc", etc., but <code>[a . c]</code> matches only "a", ".", or "c".
[]	A bracket expression. Matches a single character that is contained within the brackets. For example, <code>[abc]</code> matches "a", "b", or "c". <code>[a-z]</code> specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: <code>[abcx-z]</code> matches "a", "b", "c", "x", "y", or "z", as does <code>[a-cx-z]</code> . The - character is treated as a literal character if it is the last or the first (after the ^) character within the brackets: <code>[abc-]</code> , <code>[-abc]</code> . Note that backslash escapes are not allowed. The] character can be included in a bracket expression if it is the first (after the ^) character: <code>[] abc]</code> .
[^]	Matches a single character that is not contained within the brackets. For example, <code>[^abc]</code> matches any character other than "a", "b", or "c". <code>[^a-z]</code> matches any single character that is not a lowercase letter from "a" to "z". As above, literal characters and ranges can be mixed.
^	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
\$	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
BRE: \(\)	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, \n). A marked subexpression is also called a block or capturing group.
ERE: ()	
\n	Matches what the <i>n</i> th marked subexpression matched, where <i>n</i> is a digit from 1 to 9. This construct is theoretically irregular and was not adopted in the POSIX ERE syntax. Some tools allow referencing more than nine capturing groups.
*	Matches the preceding element zero or more times. For example, <code>ab*c</code> matches "ac", "abc", "abbbc", etc. <code>[xyz]*</code> matches "", "x", "y", "z", "zx", "zyx", "xyzy", and so on. <code>\(ab\)*</code> matches "", "ab", "abab", "ababab", and so on.
BRE: \{m, n\}	Matches the preceding element at least <i>m</i> and not more than <i>n</i> times. For example, <code>a\{3, 5\}</code> matches only "aaa", "aaaa", and "aaaaa". This is not found in a few older instances of regular expressions.
ERE: {m, n}	

Examples:

- `.at` matches any three-character string ending with "at", including "hat", "cat", and "bat".
- `[hc]at` matches "hat" and "cat".
- `[^b]at` matches all strings matched by `.at` except "bat".
- `^ [hc]at` matches "hat" and "cat", but only at the beginning of the string or line.
- `[hc]at$` matches "hat" and "cat", but only at the end of the string or line.
- `\[. \]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]" and "[b]".

POSIX Extended Regular Expressions

The meaning of metacharacters escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, `\(\)` is now `()` and `\{ \}` is now `{ }`. Additionally, support is removed for `\n` backreferences and the following metacharacters are added:

Metacharacter	Description
?	Matches the preceding element zero or one time. For example, <code>ba?</code> matches "b" or "ba".
+	Matches the preceding element one or more times. For example, <code>ba+</code> matches "ba", "baa", "baaa", and so on.
	The choice (aka alternation or set union) operator matches either the expression before or the expression after the operator. For example, <code>abc def</code> matches "abc" or "def".

Examples:

- `[hc]+at` matches "hat", "cat", "hhat", "chat", "hcat", "ccchat", and so on, but not "at".
- `[hc]?at` matches "hat", "cat", and "at".
- `[hc]*at` matches "hat", "cat", "hhat", "chat", "hcat", "ccchat", "at", and so on.
- `cat|dog` matches "cat" or "dog".

POSIX Extended Regular Expressions can often be used with modern Unix utilities by including the command line flag `-E`.

POSIX character classes

Since many ranges of characters depend on the chosen locale setting (i.e., in some settings letters are organized as `abc...zABC...Z`, while in some others as `aAbBcC...zZ`), the POSIX standard defines some classes or categories of characters as shown in the following table:

POSIX	Non-standard	Perl	ASCII	Description
<code>[:alnum:]</code>			<code>[A-Za-z0-9]</code>	Alphanumeric characters
	<code>[:word:]</code>	<code>\w</code>	<code>[A-Za-z0-9_]</code>	Alphanumeric characters plus "_"
		<code>\W</code>	<code>[^A-Za-z0-9_]</code>	Non-word characters
<code>[:alpha:]</code>			<code>[A-Za-z]</code>	Alphabetic characters
<code>[:blank:]</code>			<code>[\t]</code>	Space and tab
		<code>\b</code>	<code>[(?<=\W) (?=\w) (?<=\w) (?=\W)]</code>	Word boundaries
<code>[:cntrl:]</code>			<code>[\x00-\x1F\x7F]</code>	Control characters
<code>[:digit:]</code>		<code>\d</code>	<code>[0-9]</code>	Digits
		<code>\D</code>	<code>[^0-9]</code>	Non-digits
<code>[:graph:]</code>			<code>[\x21-\x7E]</code>	Visible characters
<code>[:lower:]</code>			<code>[a-z]</code>	Lowercase letters
<code>[:print:]</code>			<code>[\x20-\x7E]</code>	Visible characters and the space character
<code>[:punct:]</code>			<code>[\]![#\$%&'()*+,.:/;<=>?@\^_`{ }~-]</code>	Punctuation characters
<code>[:space:]</code>		<code>\s</code>	<code>[\t\r\n\f]</code>	Whitespace characters
		<code>\S</code>	<code>[^\t\r\n\f]</code>	Non-whitespace characters
<code>[:upper:]</code>			<code>[A-Z]</code>	Uppercase letters
<code>[:xdigit:]</code>			<code>[A-Fa-f0-9]</code>	Hexadecimal digits

POSIX character classes can only be used within bracket expressions. For example, `[[:upper:]ab]` matches the uppercase letters and lowercase "a" and "b".

An additional non-POSIX class understood by some tools is [:word:], which is usually defined as [:alnum:] plus underscore. This reflects the fact that in many programming languages these are the characters that may be used in identifiers. The editor Vim further distinguishes *word* and *word-head* classes (using the notation \w and \h) since in many programming languages the characters that can begin an identifier are not the same as those that can occur in other positions.

Note that what the POSIX regular expression standards call *character classes* are commonly referred to as *POSIX character classes* in other regular expression flavors which support them. With most other regular expression flavors, the term *character class* is used to describe what POSIX calls *bracket expressions*.

Perl-derivative regular expressions

Perl has a more consistent and richer syntax than the POSIX basic (BRE) and extended (ERE) regular expression standards. An example of its consistency is that \ always escapes a non-alphanumeric character. Other examples of functionality possible with Perl but not POSIX-compliant regular expressions is the concept of lazy quantification (see the next section), possessive quantifiers to control backtracking, named capture groups, and recursive patterns.

Due largely to its expressive power, many other utilities and programming languages have adopted syntax similar to Perl's — for example, Java, JavaScript, PCRE, Python, Ruby, Microsoft's .NET Framework, and the W3C's XML Schema all use regular expression syntax similar to Perl's. Some languages and tools such as Boost and PHP support multiple regular expression flavors. Perl-derivative regular expression implementations are not identical, and all implement no more than a subset of Perl's features, usually those of Perl 5.0, released in 1994. With Perl 5.10, this process has come full circle with Perl incorporating syntactic extensions originally developed in Python, PCRE, and the .NET Framework.

Simple Regular Expressions

Simple Regular Expressions is a syntax that may be used by historical versions of application programs, and may be supported within some applications for the purpose of providing backward compatibility. It is deprecated.^[10]

Lazy quantification

The standard quantifiers in regular expressions are greedy, meaning they match as much as they can, only giving back as necessary to match the remainder of the regex. For example, to find the first instance of an item between < and > symbols in this example:

```
Another whale sighting occurred on <January 26>, <2004>.
```

someone new to regexes would likely come up with the pattern <.*> or similar. However, instead of the "<January 26>" that might be expected, this pattern will actually return "<January 26>, <2004>" because the * quantifier is greedy — it will consume as many characters as possible from the input, and "January 26>, <2004" has more characters than "January 26".

Though this problem can be avoided in a number of ways (e.g., by specifying the text that is *not* to be matched: <[^>]*>), modern regular expression tools allow a quantifier to be specified as *lazy* (also known as *non-greedy*, *reluctant*, *minimal*, or *ungreedy*) by putting a question mark after the quantifier (e.g., <.*?>), or by using a modifier which reverses the greediness of quantifiers (though changing the meaning of the standard quantifiers can be confusing). By using a lazy quantifier, the expression tries the minimal match first. Though in the previous example lazy matching is used to select one of many matching results, in some cases it can also be used to improve performance when greedy matching would require more backtracking.

Patterns for non-regular languages

Many features found in modern regular expression libraries provide an expressive power that far exceeds the regular languages. For example, many implementations allow grouping subexpressions with parentheses and recalling the value they match in the same expression (**backreferences**). This means that a pattern can match strings of repeated words like "papa" or "WikiWiki", called *squares* in formal language theory. The pattern for these strings is `(. *) \1`.

The language of squares is not regular, nor is it context-free. Pattern matching with an unbounded number of back references, as supported by numerous modern tools, is NP-complete (see,^[11] Theorem 6.2).

However, many tools, libraries, and engines that provide such constructions still use the term *regular expression* for their patterns. This has led to a nomenclature where the term regular expression has different meanings in formal language theory and pattern matching. For this reason, some people have taken to using the term *regex* or simply *pattern* to describe the latter. Larry Wall, author of the Perl programming language, writes in an essay about the design of Perl 6:

“‘Regular expressions’ [...] are only marginally related to real regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I’m not going to try to fight linguistic necessity here. I will, however, generally call them “regexes” (or “regexec”, when I’m in an Anglo-Saxon mood).^[4] **”**

Fuzzy Regular Expressions

Sometimes regular expressions could be convenient for working with text in natural language, when it is necessary to take into account possible typos and spelling variants - i.e. to match text in fuzzy (approximate) way. For example, if someone wants to search for Julius Caesar in Wikipedia, he could enter in search box words like:

- Gaius Julius Caesar
- Yulius Cesar
- G. Juliy Caezar

In such cases regular expressions mechanism should implement some fuzzy string matching algorithm and, possibly, some special formula for evaluation of total "similarity" between text fragment and pattern when this pattern is complicated.

This task is closely related to both Full text search and Named entity recognition, however the former is usually aimed to apply one query against big text - whereas fuzzy regexp may be used to try many patterns against a small text fragment - and with the latter, the structural dependencies of entities can not be specified.

There are few software libraries which could be used to work with fuzzy regular expressions:

- TRE - well-developed portable free project in C, which uses syntax similar to POSIX
- FREJ - open source project in Java with non-standard syntax (which utilizes prefix, Lisp-like notation), targeted to allow easy use of substitutions of inner matched fragments in outer blocks, but lacks many features of standard regular expressions.
- agrep - old (since 1989) command-line utility (proprietary, but free for non-commercial usage).

Implementations and running times

There are at least three different algorithms that decide if and how a given regular expression matches a string.

The oldest and fastest two rely on a result in formal language theory that allows every nondeterministic finite automaton (NFA) to be transformed into a deterministic finite automaton (DFA). The DFA can be constructed explicitly and then run on the resulting input string one symbol at a time. Constructing the DFA for a regular expression of size m has the time and memory cost of $O(2^m)$, but it can be run on a string of size n in time $O(n)$. An alternative approach is to simulate the NFA directly, essentially building each DFA state on demand and then

discarding it at the next step. This keeps the DFA implicit and avoids the exponential construction cost, but running cost rises to $O(m^2n)$. The explicit approach is called the DFA algorithm and the implicit approach the NFA algorithm. Adding caching to the NFA algorithm is often called the "lazy DFA" algorithm, or just the DFA algorithm without making a distinction. These algorithms are fast, but using them for recalling grouped subexpressions, lazy quantification, and similar features is tricky.^[12] ^[13]

The third algorithm is to match the pattern against the input string by backtracking. This algorithm is commonly called NFA, but this terminology can be confusing. Its running time can be exponential, which simple implementations exhibit when matching against expressions like $(a|aa)^*b$ that contain both alternation and unbounded quantification and force the algorithm to consider an exponentially increasing number of sub-cases. This behavior can cause a security problem called Regular expression Denial of Service.

Although backtracking implementations only give an exponential guarantee in the worst case, they provide much greater flexibility and expressive power. For example, any implementation which allows the use of backreferences, or implements the various extensions introduced by Perl, must include some kind of backtracking. Some implementations try to provide the best of both algorithms by first running a fast DFA algorithm, and revert to a potentially slower backtracking algorithm only when a backreference is encountered during the match.

Unicode

In theoretical terms, any token set can be matched by regular expressions as long as it is pre-defined. In terms of historical implementations, regular expressions were originally written to use ASCII characters as their token set though regular expression libraries have supported numerous other character sets. Many modern regular expression engines offer at least some support for Unicode. In most respects it makes no difference what the character set is, but some issues do arise when extending regular expressions to support Unicode.

- Supported encoding. Some regular expression libraries expect to work on some particular encoding instead of on abstract Unicode characters. Many of these require the UTF-8 encoding, while others might expect UTF-16, or UTF-32. In contrast, Perl and Java are agnostic on encodings, instead operating on decoded characters internally.
- Supported Unicode range. Many regular expression engines support only the Basic Multilingual Plane, that is, the characters which can be encoded with only 16 bits. Currently, only a few regular expression engines (e.g., Perl's and Java's) can handle the full 21-bit Unicode range.
- Extending ASCII-oriented constructs to Unicode. For example, in ASCII-based implementations, character ranges of the form `[x-y]` are valid wherever x and y are codepoints in the range [0x00,0x7F] and $\text{codepoint}(x) \leq \text{codepoint}(y)$. The natural extension of such character ranges to Unicode would simply change the requirement that the endpoints lie in [0x00,0x7F] to the requirement that they lie in [0,0x10FFFF]. However, in practice this is often not the case. Some implementations, such as that of gawk, do not allow character ranges to cross Unicode blocks. A range like [0x61,0x7F] is valid since both endpoints fall within the Basic Latin block, as is [0x0530,0x0560] since both endpoints fall within the Armenian block, but a range like [0x0061,0x0532] is invalid since it includes multiple Unicode blocks. Other engines, such as that of the Vim editor, allow block-crossing but limit the number of characters in a range to 128.
- Case insensitivity. Some case-insensitivity flags affect only the ASCII characters. Other flags affect all characters. Some engines have two different flags, one for ASCII, the other for Unicode. Exactly which characters belong to the POSIX classes also varies.
- Cousins of case insensitivity. As ASCII has case distinction, case insensitivity became a logical feature in text searching. Unicode introduced alphabetic scripts without case like Devanagari. For these, case sensitivity is not applicable. For scripts like Chinese, another distinction seems logical: between traditional and simplified. In Arabic scripts, insensitivity to initial, medial, final, and isolated position may be desired. In Japanese, insensitivity between hiragana and katakana is sometimes useful.

- Normalization. Unicode has combining characters. Like old typewriters, plain letters can be followed by one or more non-spacing symbols (usually diacritics like accent marks) to form a single printing character, but also provides precomposed characters, i.e. characters that already include one or more combining characters. A sequence of a character + combining character should be matched with the identical single precomposed character. The process of standardizing sequences of characters + combining characters is called normalization.
- New control codes. Unicode introduced amongst others, byte order marks and text direction markers. These codes might have to be dealt with in a special way.
- Introduction of character classes for Unicode blocks, scripts, and numerous other character properties. Block properties are much less useful than script properties, because a block can have code points from several different scripts, and a script can have code points from several different blocks.^[14] In Perl and the `java.util.regex` library, properties of the form `\p{InX}` or `\p{Block=X}` match characters in block X and `\P{InX}` or `\P{Block=X}` matches code points not in that block. Similarly, `\p{Armenian}`, `\p{IsArmenian}`, or `\p{Script=Armenian}` matches any character in the Armenian script. In general, `\p{X}` matches any character with either the binary property X or the general category X. For example, `\p{Lu}`, `\p{Uppercase_Letter}`, or `\p{GC=Lu}` matches any upper-case letter. Binary properties that are *not* general categories include `\p{White_Space}`, `\p{Alphabetic}`, `\p{Math}`, and `\p{Dash}`. Examples of non-binary properties are `\p{Bidi_Class=Right_to_Left}`, `\p{Word_Break=A_Letter}`, and `\p{Numeric_Value=10}`.

Uses

Regular expressions are useful in the production of syntax highlighting systems, data validation, and many other tasks.

While regular expressions would be useful on search engines such as Google, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex. Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public. Notable exceptions: Google Code Search, Exalead.

Notes

- [1] Kleene (1956)
- [2] Raymond, Eric S. citing Dennis Ritchie (2003). "Jargon File 4.4.7: grep" (<http://catb.org/jargon/html/G/grep.html>). .
- [3] Wall, Larry and the Perl 5 development team (2006). "perlre: Perl regular expressions" (<http://perldoc.perl.org/perlre.html>). .
- [4] Wall (2002)
- [5] Hopcroft, Motwani & Ullman (2000)
- [6] Sipser (1998)
- [7] Gelade & Neven (2008)
- [8] Gruber & Holzer (2008)
- [9] Kozen (1991)
- [10] The Single Unix Specification (Version 2)
- [11] Aho (1990)
- [12] Cox (2007)
- [13] Laurikari (2009)
- [14] "UTS#18 on Unicode Regular Expressions, Annex A: Character Blocks" (http://unicode.org/reports/tr18/#Character_Blocks). .
Retrieved 2010-02-05.

References

- Aho, Alfred V. (1990). "Algorithms for finding patterns in strings". In van Leeuwen, Jan. *Handbook of Theoretical Computer Science, volume A: Algorithms and Complexity*. The MIT Press. pp. 255–300
- "Regular Expressions" (<http://www.opengroup.org/onlinepubs/007908799/xbd/re.html>). *The Single UNIX Specification, Version 2*. The Open Group. 1997
- Cox, Russ (2007). "Regular Expression Matching Can Be Simple and Fast" (<http://swtch.com/~rsc/regexp/regexp1.html>)
- Forta, Ben (2004). *Sams Teach Yourself Regular Expressions in 10 Minutes*. Sams. ISBN 0-672-32566-7.
- Friedl, Jeffrey (2002). *Mastering Regular Expressions* (<http://regex.info/>). O'Reilly. ISBN 0-596-00289-0.
- Gelade, Wouter; Neven, Frank (2008). "Succinctness of the Complement and Intersection of Regular Expressions" (<http://drops.dagstuhl.de/opus/volltexte/2008/1354>). *Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*. pp. 325–336
- Gruber, Hermann; Holzer, Markus (2008). "Finite Automata, Digraph Connectivity, and Regular Expression Size" (<http://www.hermann-gruber.com/data/icalp08.pdf>). *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)*. **5126**. pp. 39–50. doi:10.1007/978-3-540-70583-3_4
- Habibi, Mehran (2004). *Real World Regular Expressions with Java 1.4*. Springer. ISBN 1-59059-107-0.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley.
- Kleene, Stephen C. (1956). "Representation of Events in Nerve Nets and Finite Automata". In Shannon, Claude E.; McCarthy, John. *Automata Studies*. Princeton University Press. pp. 3–42
- Kozen, Dexter (1991). "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*. pp. 214–225
- Laurikari, Ville (2009). "TRE library 0.7.6" (<http://www.laurikari.net/tre/>).
- Liger, Francois; Craig McQueen, Paul Wilton (2002). *Visual Basic .NET Text Manipulation Handbook*. Wrox Press. ISBN 1-86100-730-2.
- Sipser, Michael (1998). "Chapter 1: Regular Languages". *Introduction to the Theory of Computation*. PWS Publishing. pp. 31–90. ISBN 0-534-94728-X.
- Stubblebine, Tony (2003). *Regular Expression Pocket Reference*. O'Reilly. ISBN 0-596-00415-X.
- Wall, Larry (2002). "Apocalypse 5: Pattern Matching" (<http://dev.perl.org/perl6/doc/design/apo/A05.html>).

External links

- Java Tutorials: Regular Expressions (<http://java.sun.com/docs/books/tutorial/essential/regex/index.html>)
- Perl Regular Expressions documentation (<http://perldoc.perl.org/perlre.html>)
- VBScript and Regular Expressions (<http://msdn2.microsoft.com/en-us/library/ms974570.aspx>)
- .NET Framework Regular Expressions (<http://msdn.microsoft.com/en-us/library/hs600312.aspx>)
- Regular Expressions (http://www.dmoz.org/Computers/Programming/Languages/Regular_Expressions/) at the Open Directory Project
- Pattern matching tools and libraries (<http://billposer.org/Linguistics/Computation/Resources.html#patterns>)
- Structural Regular Expressions by Rob Pike (http://doc.cat-v.org/bell_labs/structural_regexps/)
- JavaScript Regular Expressions Chapter (https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Regular_Expressions) and RegExp Object Reference (https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/RegExp) at the Mozilla Developer Center

Regular expression examples

A regular expression (also "RegEx" or "regex") is a string that is used to describe or match a set of strings according to certain syntax rules. The specific syntax rules vary depending on the specific implementation, programming language, or library in use. Additionally, the functionality of regex implementations can vary between versions.

Despite this variability, and because regular expressions can be difficult to both explain and understand without examples, this article provides a basic description of some of the properties of regular expressions by way of illustration.

Conventions

The following conventions are used in the examples.^[1]

metacharacter(s) ;;	the metacharacters column specifies the regex syntax being demonstrated
=~ m//	; indicates a regex match operation in Perl
=~ s///	; indicates a regex substitution operation in Perl

Also worth noting is that these regular expressions are all Perl-like syntax. Standard POSIX regular expressions are different.

Examples

Unless otherwise indicated, the following examples conform to the Perl programming language, release 5.8.8, January 31, 2006. This means that other implementations may lack support for some parts of the syntax shown here (e.g. basic vs. extended regex, \(\) vs. (), or lack of \d instead of POSIX [:digit:]).

The syntax and conventions used in these examples coincide with that of other programming environments as well (e.g., see *Java in a Nutshell* — Page 213, *Python Scripting for Computational Science* — Page 320, Programming PHP — Page 106).

Character(s)	Description	Example
		<p>Note that all the if statements return a TRUE value</p>
.	Normally matches any character except a newline. Within square brackets the dot is literal.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/...../) { print "\$string1 has length >= 5\n"; } </pre>
()	Groups a series of pattern elements to a single element. When you match a pattern within parentheses, you can use any of \$1, \$2, ... later to refer to the previously matched pattern.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/(H..)(o..)/) { print "We matched '\$1' and '\$2'\n"; } </pre> <p>Output:</p> <pre> We matched 'Hel' and 'o W'; </pre>

Matches the preceding pattern element one or more times.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/l+/) { print "There are one or more consecutive letter \"l\"'s in \$string1\n"; } </pre> <p>Output:</p> <pre> There are one or more consecutive letter "l"'s in Hello </pre>
Matches the preceding pattern element zero or one times.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/H.?e/) { print "There is an 'H' and a 'e' separated by "; print "0-1 characters (Ex: He Hoe)\n"; } </pre>
Modifies the *, +, or {M,N}'d regex that comes before to match as few times as possible.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/(l.+?o)/) { print "The non-greedy match with 'l' followed by one o"; print "more characters is 'llo' rather than 'llo wo'.\\n"; } </pre>
Matches the preceding pattern element zero or more times.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/e1*o/) { print "There is an 'e' followed by zero to many "; print "'l' followed by 'o' (eo, elo, ello, elllo)\n"; } </pre>
Denotes the minimum M and the maximum N match count.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/l{1,2}/) { print "There exists a substring with at least 1 "; print "and at most 2 l's in \$string1\\n"; } </pre>
Denotes a set of possible character matches.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/[aeiou]+/) { print "\$string1 contains one or more vowels.\n"; } </pre>
Separates alternate possibilities.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/(Hello Hi Pogo)/) { print "At least one of Hello, Hi, or Pogo is "; print "contained in \$string1.\n"; } </pre>

Matches a zero-width boundary between a word-class character (see next) and either a non-word class character or an edge.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/llo\b/) { print "There is a word that ends with 'llo'\n"; } </pre>
Matches an alphanumeric character, including "_" ; same as [A-Za-z0-9_] in ASCII. In Unicode [2] same as [!\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}], where the Alphabetic property contains more than just Letters, and the Decimal_Number property contains more than [0-9].	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/\w/) { print "There is at least one alphanumeric "; print "character in \$string1 (A-Z, a-z, 0-9, _) \n"; } </pre>
Matches a non -alphanumeric character, excluding "_" ; same as [^A-Za-z0-9_] in ASCII, and [^\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}] in Unicode.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/\W/) { print "The space between Hello and "; print "World is not alphanumeric\n"; } </pre>
Matches a whitespace character, which in ASCII are tab, line feed, form feed, carriage return, and space; in Unicode, also matches no-break spaces, next line, and the variable-width spaces (amongst others).	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/\s.*\s/) { print "There are TWO whitespace characters, which may "; print "be separated by other characters, in \$string1" } </pre>
Matches anything BUT a whitespace.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/\S.*\S/) { print "There are TWO non-whitespace characters, which "; print "may be separated by other characters, in \$string1" } </pre>
Matches a digit; same as [0-9] in ASCII; in Unicode, same as the \p{Digit} or \p{GC=Decimal_Number} property, which itself the same as the \p{Numeric_Type=Decimal} property.	<pre> \$string1 = "99 bottles of beer on the wall."; if (\$string1 =~ m/(\d+)/) { print "\$1 is the first number in '\$string1'\n"; } </pre> <p>Output:</p> <pre> 99 is the first number in '99 bottles of beer on the wal </pre>
Matches a non-digit; same as [^0-9] in ASCII or \P{Digit} in Unicode.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/\D/) { print "There is at least one character in \$string1"; print " that is not a digit.\n"; } </pre>

Matches the beginning of a line or string.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/^He/) { print "\$string1 starts with the characters 'He'\n"; } </pre>
Matches the end of a line or string.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/rld\$/) { print "\$string1 is a line or string "; print "that ends with 'rld'\n"; } </pre>
Matches the beginning of a string (but not an internal line).	<pre> \$string1 = "Hello\nWorld\n"; if (\$string1 =~ m/\AH/) { print "\$string1 is a string "; print "that starts with 'H'\n"; } </pre>
Matches the end of a string (but not an internal line). see Perl Best Practices — Page 240	<pre> \$string1 = "Hello\nWorld\n"; if (\$string1 =~ m/d\b\n\z/) { print "\$string1 is a string "; print "that ends with 'd'\n"; } </pre>
Matches every character except the ones inside brackets.	<pre> \$string1 = "Hello World\n"; if (\$string1 =~ m/[^abc]/) { print "\$string1 contains a character other than "; print "a, b, and c\n"; } </pre>

Notes

- [1] The character 'm' is not always required to specify a Perl match operation. For example, `m/[^abc]/` could also be rendered as `/[^abc]/`. The 'm' is only necessary if the user wishes to specify a match operation without using a forward-slash as the regex delimiter. Sometimes it is useful to specify an alternate regex delimiter in order to avoid "delimiter collision". See 'perldoc perlre (<http://perldoc.perl.org/perlre.html>)' for more details.
- [2] "UTS#18 on Unicode Regular Expressions, Annex A: Character Blocks" (http://unicode.org/reports/tr18/#Character_Blocks). . Retrieved 2010-02-05.

External links

- Test tool for php's preg_match function (http://www.santic.org/preg_match)

Finite-state machine

A **finite-state machine (FSM)** or **finite-state automaton** (plural: *automata*), or simply a **state machine**, is a behavioral model used to design computer programs. It is composed of a finite number of states associated to transitions. A transition is a set of actions that starts from one state and ends in another (or the same) state. A transition is started by a trigger, and a trigger can be an event or a condition.

Finite-state machines can model a large number of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines are sometimes used to describe neurological systems and in linguistics—to describe the grammars of natural languages.

Concepts and vocabulary

A *state* describes a behavioral node of the system in which it is waiting for a trigger to execute a transition. Typically a state is introduced when the system does not react the same way to the same trigger. In the example of a car radio system, when listening to the radio (in the radio state), the "next" stimulus means going to the next station. But when the system is in the CD state, the "next" stimulus means going to the next track. The same stimulus triggers different actions depending on the current state. In some Finite-state machine representations, it is also possible to associate actions to a state:

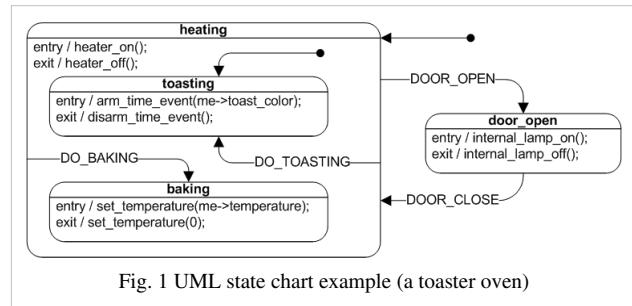
- Entry action: which is performed *when entering* the state,
- Exit action: which is performed *when exiting* the state.

A *transition* is a set of actions to be executed when a condition is fulfilled or when an event is received.

Representations

State/Event table

Besides this, several state transition table types are used. The most common representation is shown below: the combination of current state (e.g. B) and input (e.g. Y) shows the next state (e.g. C). The complete actions information is not directly described in the table, can be added only using footnotes. An FSM definition including the full actions information is possible using state tables (see also VFSM).



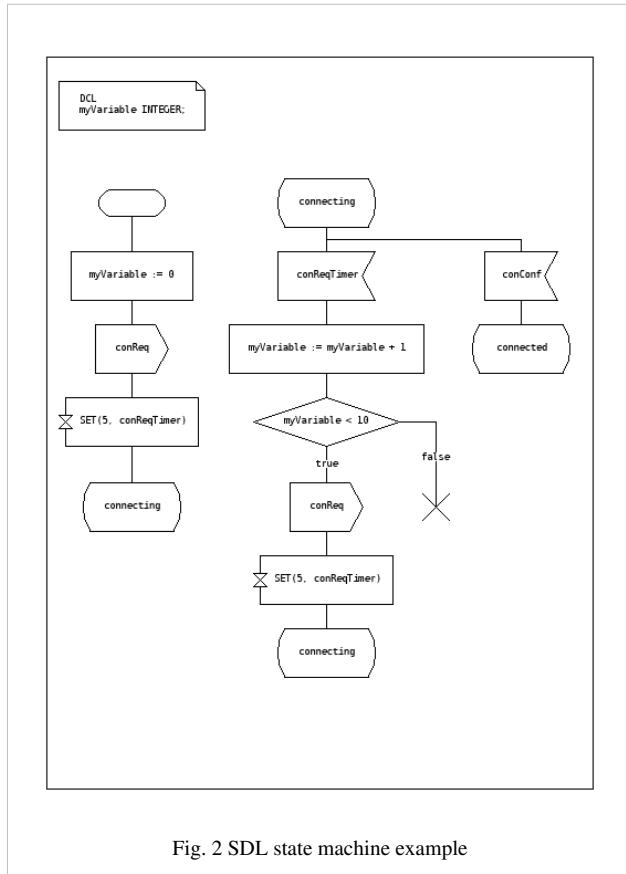


Fig. 2 SDL state machine example

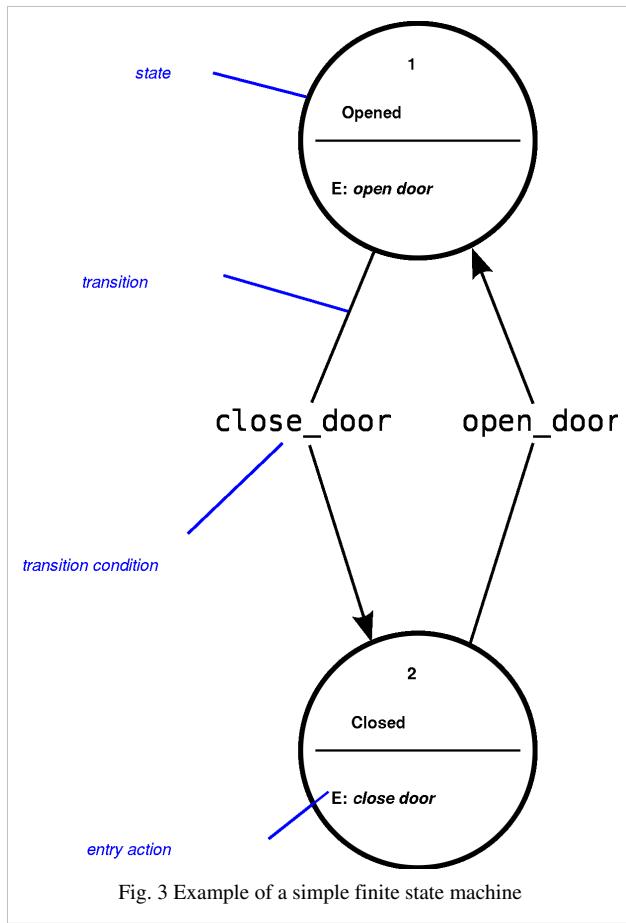


Fig. 3 Example of a simple finite state machine

State transition table

Current state → Input ↓	State A	State B	State C
Input X
Input Y	...	State C	...
Input Z

UML state machines

The Unified Modeling Language has a notation for describing state machines. UML state machines overcome the limitations of traditional finite state machines while retaining their main benefits. UML state machines introduce the new concepts of hierarchically nested states and orthogonal regions, while extending the notion of actions. UML state machines have the characteristics of both Mealy machines and Moore machines. They support actions that depend on both the state of the system and the triggering event, as in Mealy machines, as well as entry and exit actions, which are associated with states rather than transitions, as in Moore machines.

SDL state machines

The Specification and Description Language is a standard from ITU and is one of the best languages to describe state machines because it includes graphical symbols to describe actions in the transition:

- send an event
- receive an event
- start a timer
- cancel a timer
- start another concurrent state machine
- decision

SDL embeds basic data types called Abstract Data Types, an action language, and an execution semantic in order to make the finite state machine executable.

Other state diagrams

There are a large number of variants to represent an FSM such as the one in figure 3.

Usage

In addition to their use in modeling reactive systems presented here, finite state automata are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. Finite state machines are a class of automata studied in automata theory and the theory of computation. In computer science, finite state machines are widely used in modeling of application behavior, design of hardware digital systems, software engineering, compilers, network protocols, and the study of computation and languages.

Classification

There are two different groups of state machines: Acceptors/Recognizers and Transducers.

Acceptors and recognizers

Acceptors and **recognizers** (also **sequence detectors**) produce a binary output, saying either *yes* or *no* to answer whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. At the time when all input is processed, if the current state is an accepting state, the input is accepted; otherwise it is rejected. As a rule the input are symbols (characters); actions are not used. The example in figure 4 shows a finite state machine which accepts the word "nice". In this FSM the only accepting state is number 7.

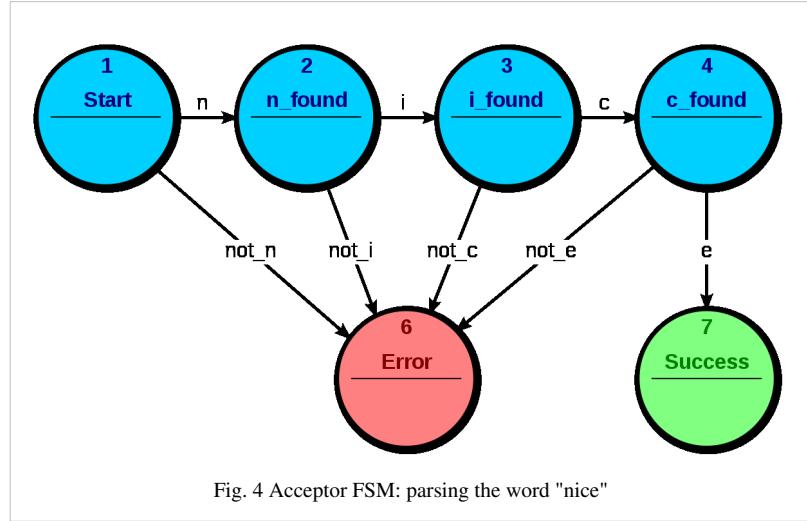


Fig. 4 Acceptor FSM: parsing the word "nice"

The machine can also be described as defining a language, which would contain every word accepted by the machine but none of the rejected ones; we say then that the language is *accepted* by the machine. By definition, the languages accepted by FSMs are the regular languages—that is, a language is regular if there is some FSM that accepts it.

Start state

The start state is usually shown drawn with an arrow "pointing at it from anywhere" (Sipser (2006) p. 34).

Accept (or final) states

Accept states (also referred to as **accepting** or **final** states) are those at which the machine reports that the input string, as processed so far, is a member of the language it accepts. It is usually represented by a double circle.

An example of an accepting state appears in the diagram to the right: a deterministic finite automaton (DFA) that detects whether the binary input string contains an even number of 0's.

S_1 (which is also the start state) indicates the state at which an even number of 0's has been input. S_1 is therefore an accepting state. This machine will finish in an accept state, if the binary string contains an even number of 0's (including any binary string containing no 0's). Examples of strings accepted by this DFA are epsilon (the empty string), 1, 11, 11..., 00, 010, 1010, 10110, etc...

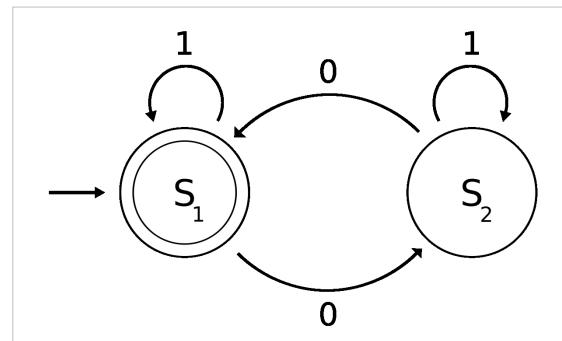


Fig. 5: Representation of a finite-state machine; this example shows one that determines whether a binary number has an odd or even number of 0's, where S_1 is an **accepting state**.

Transducers

Transducers generate output based on a given input and/or a state using actions. They are used for control applications and in the field of computational linguistics. Here two types are distinguished:

Moore machine

The FSM uses only entry actions, i.e., output depends only on the state. The advantage of the Moore model is a simplification of the behaviour. Consider an elevator door. The state machine recognizes two commands: "command_open" and "command_close" which trigger state changes. The entry action (E:) in state "Opening" starts a motor opening the door, the entry action in state "Closing" starts a motor in the other direction closing the door. States "Opened" and "Closed" stop the motor when fully opened or closed. They signal to the outside world (e.g., to other state machines) the situation: "door is open" or "door is closed".

Mealy machine

The FSM uses only input actions, i.e., output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states. The example in figure 7 shows a Mealy FSM implementing the same behaviour as in the Moore example (the behaviour depends on the implemented FSM execution model and will work, e.g., for virtual FSM but not for event driven FSM). There are two input actions (I:): "start motor to close the door if command_close arrives" and "start motor in the other direction to open the door if command_open arrives". The "opening" and "closing" intermediate states are not shown.

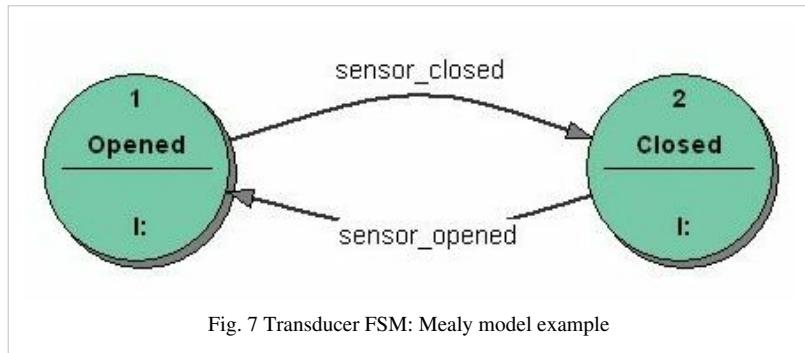


Fig. 7 Transducer FSM: Mealy model example

In practice^[of what?] mixed models are often used.

More details about the differences and usage of Moore and Mealy models, including an executable example, can be found in the external technical note "Moore or Mealy model?"^[1]

Determinism

A further distinction is between **deterministic** (DFA) and **non-deterministic** (NFA, GNFA) automata. In deterministic automata, every state has exactly one transition for each possible input. In non-deterministic automata, an input can lead to one, more than one or no transition for a given state. This distinction is relevant in practice, but not in theory, as there exists an algorithm (the powerset construction) which can transform any NFA into a more complex DFA with identical functionality.

The FSM with only one state is called a combinatorial FSM and uses only input actions. This concept is useful in cases where a number of FSM are required to work together, and where it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools.

Alternative semantics

There are other sets of semantics available to represent state machines. For example, there are tools for modeling and designing logic for embedded controllers.^[2] They combine hierarchical state machines, flow graphs, and truth tables into one language, resulting in a different formalism and set of semantics.^[3] Figure 8 illustrates this mix of state machines and flow graphs with a set of states to represent the state of a stopwatch and a flow graph to control the ticks of the watch. These charts, like Harel's original state machines,^[4] support hierarchically nested states,

orthogonal regions, state actions, and transition actions.^[5]

FSM logic

The next state and output of an FSM is a function of the input and of the current state. The FSM logic is shown in Figure 8.

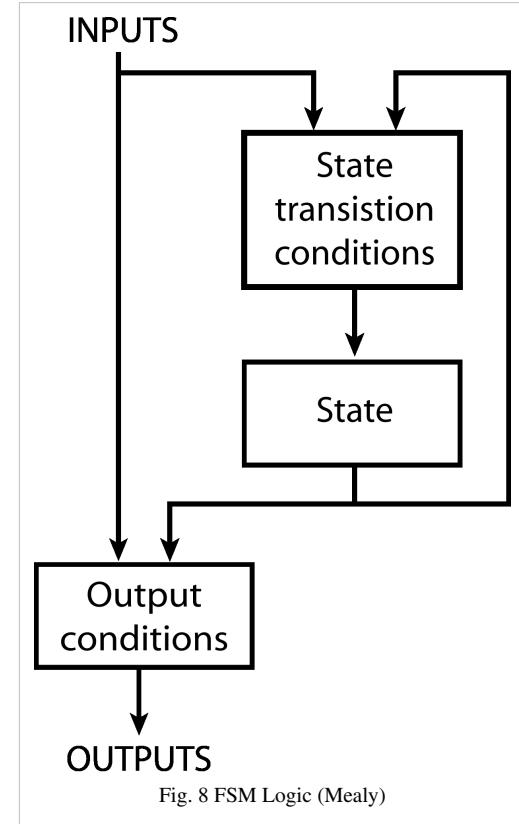


Fig. 8 FSM Logic (Mealy)

Mathematical model

In accordance with the general classification, the following formal definitions are found:

- A *deterministic finite state machine* or *acceptor deterministic finite state machine* is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:
 - Σ is the input alphabet (a finite, non-empty set of symbols).
 - S is a finite, non-empty set of states.
 - s_0 is an initial state, an element of S .
 - δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$ (in a nondeterministic finite state machine it would be $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, i.e., δ would return a set of states).
 - F is the set of final states, a (possibly empty) subset of S .

For both deterministic and non-deterministic FSMs, it is conventional to allow δ to be a partial function, i.e. $\delta(q, x)$ does not have to be defined for every combination of $q \in S$ and $x \in \Sigma$. If an FSM M is in a state q , the next symbol is x and $\delta(q, x)$ is not defined, then M can announce an error (i.e. reject the input). This is useful in definitions of general state machines, but less useful when transforming the machine. Some algorithms in their default form may require total functions.

- A *finite state transducer* is a sextuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$, where:
 - Σ is the input alphabet (a finite non empty set of symbols).
 - Γ is the output alphabet (a finite, non-empty set of symbols).
 - S is a finite, non-empty set of states.

- s_0 is the initial state, an element of S . In a nondeterministic finite state machine, s_0 is a set of initial states.
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$.
- ω is the output function.

If the output function is a function of a state and input alphabet ($\omega : S \times \Sigma \rightarrow \Gamma$) that definition corresponds to the **Mealy model**, and can be modelled as a Mealy machine. If the output function depends only on a state ($\omega : S \rightarrow \Gamma$) that definition corresponds to the **Moore model**, and can be modelled as a Moore machine. A finite-state machine with no output function at all is known as a semiautomaton or transition system.

If we disregard the first output symbol of a Moore machine, $\omega(s_0)$, then it can be readily converted to an output-equivalent Mealy machine by setting the output function of every Mealy transition (i.e. labeling every edge) with the output symbol given of the destination Moore state. The converse transformation is less straightforward because a Mealy machine state may have different output labels on its incoming transitions (edges). Every such state needs to be split in multiple Moore machine states, one for every incident output symbol.^[6]

Optimization

Optimizing an FSM means finding the machine with the minimum number of states that performs the same function. The fastest known algorithm doing this is the Hopcroft minimization algorithm.^[7] ^[8] Other techniques include using an implication table, or the Moore reduction procedure. Additionally, acyclic FSAs can be optimized using a simple bottom up algorithm.

Implementation

Hardware applications

In a digital circuit, an FSM may be built using a programmable logic device, a programmable logic controller, logic gates and flip flops or relays. More specifically, a hardware implementation requires a register to store state variables, a block of combinational logic which determines the state transition, and a second block of combinational logic that determines the output of an FSM. One of the classic hardware implementations is the Richards controller.

Mealy and Moore machines produce logic with asynchronous output, because there is a propagation delay between the flip-flop and output. This causes slower operating frequencies in FSM. A Mealy or Moore machine can be convertible to a FSM which output is directly from a flip-flop, which makes the FSM run at higher frequencies. This kind of FSM is sometimes called Medvedev FSM.^[9] A counter is the simplest form of this kind of FSM.

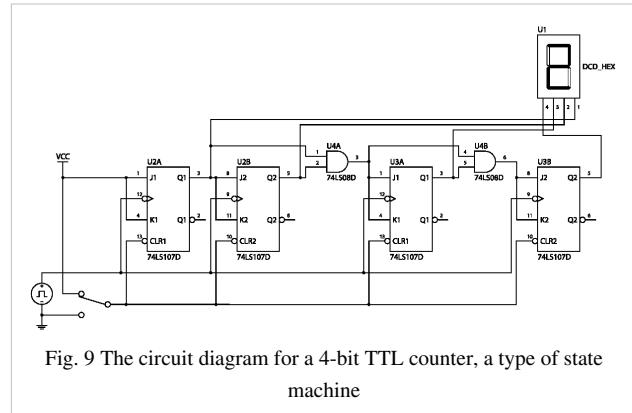


Fig. 9 The circuit diagram for a 4-bit TTL counter, a type of state machine

Software applications

The following concepts are commonly used to build software applications with finite state machines:

- Automata-based programming
- Event driven FSM
- Virtual FSM (VFSM)

References

- [1] <http://www.stateworks.com/technology/TN10-Moore-Or-Mealy-Model/>
- [2] Tiwari, A. (2002). Formal Semantics and Analysis Methods for Simulink Stateflow Models. (<http://www.csl.sri.com/users/tiwari/papers/stateflow.pdf>)
- [3] Hamon, G. (2005). A Denotational Semantics for Stateflow. International Conference on Embedded Software (pp. 164–172). Jersey City, NJ: ACM. (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.8817&rep=rep1&type=pdf>)
- [4] Harel, D. (1987). A Visual Formalism for Complex Systems. *Science of Computer Programming*, 231–274. (<http://www.fceia.unr.edu.ar/asist/harel01.pdf>)
- [5] Alur, R., Kanade, A., Ramesh, S., & Shashidhar, K. C. (2008). Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. Internation Conference on Embedded Software (pp. 89–98). Atlanta, GA: ACM. (http://drona.csa.iisc.ernet.in/~kanade/publications/symbolic_analysis_for_improving_simulation_coverage_of_simulink_stateflow_models.pdf)
- [6] James Andrew Anderson; Thomas J. Head (2006). *Automata theory with modern applications* (<http://books.google.com/books?id=ikS8BLdLDxIC&pg=PA105>). Cambridge University Press. pp. 105–108. ISBN 9780521848879. .
- [7] Hopcroft, John E (1971). An $n \log n$ algorithm for minimizing states in a finite automaton (<ftp://reports.stanford.edu/pub/cstr/reports/cs/tr71/190/CS-TR-71-190.pdf>)
- [8] Almeida, Marco; Moreira, Nelma; Reis, Rogerio (2007). On the performance of automata minimization algorithms. (<http://www.dcc.fc.up.pt/dcc/Pubs/TReports/TR07/dcc-2007-03.pdf>)
- [9] "FSM: Medvedev" (http://www.vhdl-online.de/tutorial/deutsch/ct_226.htm). .

Further reading

General

- Wagner, F., "Modeling Software with Finite State Machines: A Practical Approach", Auerbach Publications, 2006, ISBN 0-8493-8086-3.
- ITU-T, *Recommendation Z.100 Specification and Description Language (SDL)* (<http://www.itu.int/rec/T-REC-Z.100-200711-I/en>)
- Samek, M., *Practical Statecharts in C/C++* (<http://www.state-machine.com/psicc/index.php>), CMP Books, 2002, ISBN 1-57820-110-1.
- Samek, M., *Practical UML Statecharts in C/C++, 2nd Edition* (<http://www.state-machine.com/psicc2/index.php>), Newnes, 2008, ISBN 0-7506-8706-1.
- Gardner, T., *Advanced State Management* (<http://www.troyworks.com/cogs/>), 2007
- Cassandras, C., Lafontaine, S., "Introduction to Discrete Event Systems". Kluwer, 1999, ISBN 0-7923-8609-4.
- Timothy Kam, *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Publishers, Boston 1997, ISBN 0-7923-9842-4
- Tiziano Villa, *Synthesis of Finite State Machines: Logic Optimization*. Kluwer Academic Publishers, Boston 1997, ISBN 0-7923-9892-0
- Carroll, J., Long, D., *Theory of Finite Automata with an Introduction to Formal Languages*. Prentice Hall, Englewood Cliffs, 1989.
- Kohavi, Z., *Switching and Finite Automata Theory*. McGraw-Hill, 1978.
- Gill, A., *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.
- Ginsburg, S., *An Introduction to Mathematical Machine Theory*. Addison-Wesley, 1962.

Finite state machines (automata theory) in theoretical computer science

- Arbib, Michael A. (1969). *Theories of Abstract Automata* (1st ed.). Englewood Cliffs, N.J.: Prentice-Hall, Inc.. ISBN 0139133682.
- Bobrow, Leonard S.; Michael A. Arbib (1974). *Discrete Mathematics: Applied Algebra for Computer and Information Science* (1st ed.). Philadelphia: W. B. Saunders Company, Inc.. ISBN 0721617689.
- Booth, Taylor L. (1967). *Sequential Machines and Automata Theory* (1st ed.). New York: John Wiley and Sons, Inc.. Library of Congress Card Catalog Number 67-25924. Extensive, wide-ranging book meant for specialists, written for both theoretical computer scientists as well as electrical engineers. With detailed explanations of state minimization techniques, FSMs, Turing machines, Markov processes, and undecidability. Excellent treatment of Markov processes.
- Boolos, George; Richard Jeffrey (1989, 1999). *Computability and Logic* (3rd ed.). Cambridge, England: Cambridge University Press. ISBN 0-521-20402-X. Excellent. Has been in print in various editions and reprints since 1974 (1974, 1980, 1989, 1999).
- Brookshear, J. Glenn (1989). *Theory of Computation: Formal Languages, Automata, and Complexity*. Redwood City, California: Benjamin/Cummings Publish Company, Inc.. ISBN 0-8053-0143-7. Approaches Church-Turing thesis from three angles: levels of finite automata as acceptors of formal languages, primitive and partial recursive theory, and power of bare-bones programming languages to implement algorithms, all in one slim volume.
- Davis, Martin; Ron Sigal, Elaine J. Weyuker (1994). *Computability, Complexity, and Languages and Logic: Fundamentals of Theoretical Computer Science* (2nd ed.). San Diego: Academic Press, Harcourt, Brace & Company. ISBN 0122063821.
- Hopcroft, John; Jeffrey Ullman (1979). *Introduction to Automata Theory, Languages, and Computation* (1st ed.). Reading Mass: Addison-Wesley. ISBN 0-201-02988-X. An excellent book centered around the issues of machine-interpretation of "languages", NP-Completeness, etc.
- Hopcroft, John E.; Rajeev Motwani, Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Reading Mass: Addison-Wesley. ISBN 0201441241. Distinctly different and less intimidating than the first edition.
- Hopkin, David; Barbara Moss (1976). *Automata*. New York: Elsevier North-Holland. ISBN 0-444-00249-9.
- Kozen, Dexter C. (1997). *Automata and Computability* (1st ed.). New York: Springer-Verlag. ISBN 0-387-94907-0.
- Lewis, Harry R.; Christos H. Papadimitriou (1998). *Elements of the Theory of Computation* (2nd ed.). Upper Saddle River, New Jersey: Prentice-Hall. ISBN 0-13-262478-8.
- Linz, Peter (2006). *Formal Languages and Automata* (4th ed.). Sudbury, MA: Jones and Bartlett. ISBN 978-0-7637-3798-6.
- Minsky, Marvin (1967). *Computation: Finite and Infinite Machines* (1st ed.). New Jersey: Prentice-Hall. Minsky spends pages 11–20 defining what a "state" is in context of FSMs. His state diagram convention is unconventional. Excellent, i.e., relatively readable, sometimes funny.
- Christos Papadimitriou (1993). *Computational Complexity* (1st ed.). Addison Wesley. ISBN 0-201-53082-1.
- Pippenger, Nicholas (1997). *Theories of Computability* (1st ed.). Cambridge, England: Cambridge University Press. ISBN 0-521-55380-6 (hc). Abstract algebra is at the core of the book, rendering it advanced and less accessible than other texts.
- Rodger, Susan; Thomas Finley (2006). *JFLAP: An Interactive Formal Languages and Automata Package* (1st ed.). Sudbury, MA: Jones and Bartlett. ISBN 0-7637-3834-4.
- Sipser, Michael (2006). *Introduction to the Theory of Computation* (2nd ed.). Boston Mass: Thomson Course Technology. ISBN 0-534-95097-3. cf Finite state machines (finite automata) in chapter 29.
- Wood, Derick (1987). *Theory of Computation* (1st ed.). New York: Harper & Row, Publishers, Inc.. ISBN 0-06-047208-1.

Abstract state machines in theoretical computer science

- Yuri Gurevich (2000). *Sequential Abstract State Machines Capture Sequential Algorithms*, ACM Transactions on Computational Logic, v1. 1, no. 1 (July 2000), pages 77–111. <http://research.microsoft.com/~gurevich/Opera/141.pdf>

Machine learning using finite-state algorithms

- Mitchell, Tom M. (1997). *Machine Learning* (1st ed.). New York: WCB/McGraw-Hill Corporation. ISBN 0-07-042807-7. A broad brush but quite thorough and sometimes difficult, meant for computer scientists and engineers. Chapter 13 *Reinforcement Learning* deals with robot-learning involving state-machine-like algorithms.

Hardware engineering: state minimization and synthesis of sequential circuits

- Booth, Taylor L. (1967). *Sequential Machines and Automata Theory* (1st ed.). New York: John Wiley and Sons, Inc.. Library of Congress Card Catalog Number 67-25924. Extensive, wide-ranging book meant for specialists, written for both theoretical computer scientists as well as electrical engineers. With detailed explanations of state minimization techniques, FSMs, Turing machines, Markov processes, and undecidability. Excellent treatment of Markov processes.
- Booth, Taylor L. (1971). *Digital Networks and Computer Systems* (1st ed.). New York: John Wiley and Sons, Inc.. ISBN 0-471-08840-4. Meant for electrical engineers. More focused, less demanding than his earlier book. His treatment of computers is out-dated. Interesting take on definition of "algorithm".
- McCluskey, E. J. (1965). *Introduction to the Theory of Switching Circuits* (1st ed.). New York: McGraw-Hill Book Company, Inc.. Library of Congress Card Catalog Number 65-17394. Meant for hardware electrical engineers. With detailed explanations of state minimization techniques and synthesis techniques for design of combinatory logic circuits.
- Hill, Fredrick J.; Gerald R. Peterson (1965). *Introduction to the Theory of Switching Circuits* (1st ed.). New York: McGraw-Hill Book Company. Library of Congress Card Catalog Number 65-17394. Meant for hardware electrical engineers. Excellent explanations of state minimization techniques and synthesis techniques for design of combinatory and sequential logic circuits.

Finite Markov chain processes

"We may think of a Markov chain as a process that moves successively through a set of states s_1, s_2, \dots, s_r if it is in state s_i it moves on to the next stop to state s_j with probability p_{ij} . These probabilities can be exhibited in the form of a transition matrix" (Kemeny (1959), p. 384)

Finite Markov-chain processes are also known as subshifts of finite type.

- Booth, Taylor L. (1967). *Sequential Machines and Automata Theory* (1st ed.). New York: John Wiley and Sons, Inc.. Library of Congress Card Catalog Number 67-25924. Extensive, wide-ranging book meant for specialists, written for both theoretical computer scientists as well as electrical engineers. With detailed explanations of state minimization techniques, FSMs, Turing machines, Markov processes, and undecidability. Excellent treatment of Markov processes.
- Kemeny, John G.; Hazleton Mirkil, J. Laurie Snell, Gerald L. Thompson (1959). *Finite Mathematical Structures* (1st ed.). Englewood Cliffs, N.J.: Prentice-Hall, Inc.. Library of Congress Card Catalog Number 59-12841. Classical text. cf. Chapter 6 "Finite Markov Chains".

External links

- Free On-Line Dictionary of Computing (<http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=finite+state+machine>) description of Finite State Machines
- NIST Dictionary of Algorithms and Data Structures (<http://www.nist.gov/dads/HTML/finiteStateMachine.html>) description of Finite State Machines

Preprocessor

In computer science, a **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers. The amount and kind of processing done depends on the nature of the preprocessor; some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of full-fledged programming languages.

A common example from computer programming is the processing performed on source code before the next step of compilation. In some computer languages (e.g., C and PL/I) there is a phase of translation known as *preprocessing*.

Lexical preprocessors

Lexical preprocessors are the lowest-level of preprocessors, in so far as they only require lexical analysis, that is, they operate on the source text, prior to any parsing, by performing simple substitution of tokenized character sequences for other tokenized character sequences, according to user-defined rules. They typically perform macro substitution, textual inclusion of other files, and conditional compilation or inclusion.

C preprocessor

The most common example of this is the C preprocessor, which takes lines beginning with '#' as directives. Because it knows nothing about the underlying language, its use has been criticized and many of its features built directly into other languages. For example, macros replaced with aggressive inlining and templates, includes with compile-time imports (this requires the preservation of type information in the object code, making this feature impossible to retrofit into a language); conditional compilation is effectively accomplished with `if-then-else` and dead code elimination in some languages.

Other lexical preprocessors

Other lexical preprocessors include the general-purpose m4, most commonly used in cross-platform build systems such as autoconf, and GEMA, an open source macro processor which operates on patterns of context.

Syntactic preprocessors

Syntactic preprocessors were introduced with the Lisp family of languages. Their role is to transform syntax trees according to a number of user-defined rules. For some programming languages, the rules are written in the same language as the program (compile-time reflection). This is the case with Lisp and OCaml. Some other languages rely on a fully external language to define the transformations, such as the XSLT preprocessor for XML, or its statically typed counterpart CDuce.

Syntactic preprocessors are typically used to customize the syntax of a language, extend a language by adding new primitives, or embed a Domain-Specific Programming Language inside a general purpose language.

Customizing syntax

A good example of syntax customization is the existence of two different syntaxes in the Objective Caml programming language.^[1] Programs may be written indifferently using the "normal syntax" or the "revised syntax", and may be pretty-printed with either syntax on demand.

Similarly, a number of programs written in OCaml customize the syntax of the language by the addition of new operators.

Extending a language

The best examples of language extension through macros are found in the Lisp family of languages. While the languages, by themselves, are simple dynamically typed functional cores, the standard distributions of Scheme or Common Lisp permit imperative or object-oriented programming, as well as static typing. Almost all of these features are implemented by syntactic preprocessing, although it bears noting that the "macro expansion" phase of compilation is handled by the compiler in Lisp. This can still be considered a form of preprocessing, since it takes place before other phases of compilation.

Similarly, statically checked, type-safe regular expressions or code generation may be added to the syntax and semantics of OCaml through macros, as well as micro-threads (also known as coroutines or fibers), monads or transparent XML manipulation.

Specializing a language

One of the unusual features of the Lisp family of languages is the possibility of using macros to create an internal Domain-Specific Programming Language. Typically, in a large Lisp-based project, a module may be written in a variety of such minilanguages, one perhaps using a SQL-based dialect of Lisp, another written in a dialect specialized for GUIs or pretty-printing, etc. Common Lisp's standard library contains an example of this level of syntactic abstraction in the form of the LOOP macro, which implements an Algol-like minilanguage to describe complex iteration, while still enabling the use of standard Lisp operators.

The MetaOCaml preprocessor/language provides similar features for external Domain-Specific Programming Languages. This preprocessor takes the description of the semantics of a language (i.e. an interpreter) and, by combining compile-time interpretation and code generation, turns that definition into a compiler to the OCaml programming language—and from that language, either to bytecode or to native code.

General purpose preprocessor

Most preprocessors are specific to a particular data processing task (e.g., compiling the C language). A preprocessor may be promoted as being *general purpose*, meaning that it is not aimed at a specific usage or programming language, and is intended to be used for a wide variety of text processing tasks.

M4 is probably the most well known example of such a general purpose preprocessor, although the C preprocessor is sometimes used in a non-C specific role. Examples:

- using C preprocessor for Javascript preprocessing.^[2]
- using M4 (see on-article example) or C preprocessor^[3] as a template engine, to HTML generation.
- imake, a make interface using the C preprocessor, used in the X Window System but now deprecated in favour of automake.
- grompp, a preprocessor for simulation input files for GROMACS (a fast, free, open-source code for some problems in computational chemistry) which calls the system C preprocessor (or other preprocessor as determined by the simulation input file) to parse the topology, using mostly the #define and #include mechanisms to determine the effective topology at grompp run time.

References

- [1] The Revised syntax (<http://caml.inria.fr/pub/docs/manual-camlp4/manual007.html>) from The Caml language website
- [2] Show how to use C-preprocessor on JavaScript files. "JavaScript is Not Industrial Strength" (<http://web.archive.org/web/20080116034428/http://blog.inetoffice.com/?p=12>) by *T. Snyder*.
- [3] Show how to use C-preprocessor as template engine. "Using a C preprocessor as an HTML authoring tool" (<http://www.cs.tut.fi/~jkorpela/html/cpre.html>) by *J. Korpela*, 2000.

External links

- DSL Design in Lisp (<http://lispm.dyndns.org/news?ID=NEWS-2005-07-08-1>)
- Programming from the bottom up (<http://www.paulgraham.com/progbot.html>)
- The Generic PreProcessor (<http://www.owl-s.org/tools.en.html#gpp>)
- Gema, the General Purpose Macro Processor (<http://gema.sourceforge.net>)
- The PIKT piktc (<http://pikt.org/pikt/ref/ref.4.piktc.html>) text, script, and configuration file preprocessor (<http://pikt.org/pikt/ref/ref.3.html>)
- minimac, a minimalist macro processor (<http://freshmeat.net/projects/minimac-macro-processor>)
- Java Comment Preprocessor (<http://igormaznitsa.com/projects/jcp/index.html>)

Syntactic analysis

Parsing

In computer science and linguistics, **parsing**, or, more formally, **syntactic analysis**, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Parsing can also be used as a linguistic term, especially in reference to how phrases are divided up in garden path sentences.

Parsing is also an earlier term for the diagramming of sentences of natural languages, and is still used for the diagramming of inflected languages, such as the Romance languages or Latin. The term parsing comes from Latin *pars* (*ōrātiōnis*), meaning part (of speech).^[1] ^[2]

Parsing is a common term used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings, rather than computers, analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc."^[3] This term is especially common when discussing what linguistic cues help speakers to parse garden-path sentences.

Parser

In computing, a **parser** is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analyser to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool.

Human languages

In some machine translation and natural language processing systems, human languages are parsed by computer programs. Human sentences are not easily parsed by programs, as there is substantial ambiguity in the structure of human language, whose usage is to convey meaning (or semantics) amongst a potentially unlimited range of possibilities but only some of which are germane to the particular case. So an utterance "Man bites dog" versus "Dog bites man" is definite on one detail but in another language might appear as "Man dog is biting" with a reliance on the larger context to distinguish between those two possibilities, if indeed that difference was of concern. It is difficult to prepare formal rules to describe informal behaviour even though it is clear that some rules are being followed.

In order to parse natural language data, researchers must first agree on the grammar to be used. The choice of syntax is affected by both linguistic and computational concerns; for instance some parsing systems use lexical functional grammar, but in general, parsing for grammars of this type is known to be NP-complete. Head-driven phrase structure grammar is another linguistic formalism which has been popular in the parsing community, but other research efforts have focused on less complex formalisms such as the one used in the Penn Treebank. Shallow parsing aims to find only the boundaries of major constituents such as noun phrases. Another popular strategy for avoiding linguistic controversy is dependency grammar parsing.

Most modern parsers are at least partly statistical; that is, they rely on a corpus of training data which has already been annotated (parsed by hand). This approach allows the system to gather information about the frequency with which various constructions occur in specific contexts. (See *machine learning*.) Approaches which have been used

include straightforward PCFGs (probabilistic context free grammars), maximum entropy, and neural nets. Most of the more successful systems use *lexical* statistics (that is, they consider the identities of the words involved, as well as their part of speech). However such systems are vulnerable to overfitting and require some kind of smoothing to be effective.

Parsing algorithms for natural language cannot rely on the grammar having 'nice' properties as with manually-designed grammars for programming languages. As mentioned earlier some grammar formalisms are very difficult to parse computationally; in general, even if the desired structure is not context-free, some kind of context-free approximation to the grammar is used to perform a first pass. Algorithms which use context-free grammars often rely on some variant of the CKY algorithm, usually with some heuristic to prune away unlikely analyses to save time. (*See chart parsing.*) However some systems trade speed for accuracy using, e.g., linear-time versions of the shift-reduce algorithm. A somewhat recent development has been parse reranking in which the parser proposes some large number of analyses, and a more complex system selects the best option.

Programming languages

The most common use of a parser is as a component of a compiler or interpreter. This parses the source code of a computer programming language to create some form of internal representation. Programming languages tend to be specified in terms of a context-free grammar because fast and efficient parsers can be written for them. Parsers are written by hand or generated by parser generators.

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out.

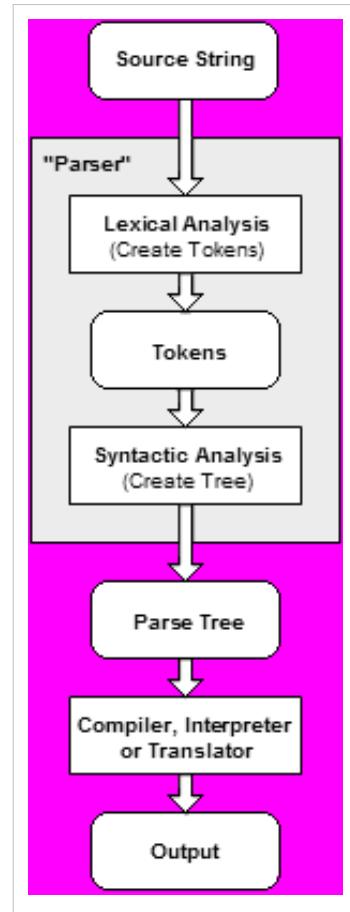
Overview of process

The following example demonstrates the common case of parsing a computer language with two levels of grammar: lexical and syntactic.

The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. For example, a calculator program would look at an input such as "12 * (3+4) ^ 2" and split it into the tokens 12, *, (, 3, +, 4,), ^, and 2, each of which is a meaningful symbol in the context of an arithmetic expression. The lexer would contain rules to tell it that the characters *, +, ^, (and) mark the start of a new token, so meaningless tokens like "12*" or "(3" will not be generated.

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars.

The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. In the case of a calculator or interpreter, the action is to evaluate the expression or program; a compiler, on the other hand, would generate some kind of code. Attribute grammars can also be used to define these actions.



Types of parser

The *task* of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- Top-down parsing- Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.^[4]
- Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

LL parsers and recursive-descent parser are examples of top-down parsers which cannot accommodate left recursive productions. Although it has been believed that simple implementations of top-down parsing cannot accommodate direct and indirect left-recursion and may require exponential time and space complexity while parsing ambiguous context-free grammars, more sophisticated algorithms for top-down parsing have been created by Frost, Hafiz, and Callaghan^[5] ^[6] which accommodate ambiguity and left recursion in polynomial time and which generate polynomial-size representations of the potentially-exponential number of parse trees. Their algorithm is able to produce both left-most and right-most derivations of an input with regard to a given CFG.

An important distinction with regard to parsers is whether a parser generates a *leftmost derivation* or a *rightmost derivation* (see context-free grammar). LL parsers will generate a leftmost derivation and LR parsers will generate a rightmost derivation (although usually in reverse).^[4]

Examples of parsers

Top-down parsers

Some of the parsers that use top-down parsing include:

- Recursive descent parser
- LL parser (**L**evel-to-right, **L**eftmost derivation)
- Earley parser
- X-SAIGA^[7] - eXecutable SpecificAtIons of GrAmmars. Contains publications related to top-down parsing algorithm that supports left-recursion and ambiguity in polynomial time and space.

Bottom-up parsers

Some of the parsers that use bottom-up parsing include:

- Precedence parser
 - Operator-precedence parser
 - Simple precedence parser
- BC (bounded context) parsing
- LR parser (**L**evel-to-right, **R**ightmost derivation)
 - Simple LR (SLR) parser
 - LALR parser
 - Canonical LR (LR(1)) parser
 - GLR parser
- CYK parser

Parser development software

Some of the well known parser development tools include the following. Also see comparison of parser generators.

- ANTLR
- Bison
- Coco/R
- GOLD
- JavaCC
- Lemon
- Lex
- Parboiled
- ParseIT
- Ragel
- Syntax Definition Formalism
- Spirit Parser Framework
- SYNTAX
- Yacc

References

- [1] "Bartleby.com homepage" (<http://www.bartleby.com/61/33/P0083300.html>). . Retrieved 28 November 2010.
- [2] "parse" (<http://dictionary.reference.com/search?q=parse&x=0&y=0>). dictionary.reference.com. . Retrieved 27 November 2010.
- [3] "parse" (<http://dictionary.reference.com/browse/parse>). dictionary.reference.com. . Retrieved 27 November 2010.
- [4] Aho, A.V., Sethi, R. and Ullman J.D. (1986) " Compilers: principles, techniques, and tools." *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.*
- [5] Frost, R., Hafiz, R. and Callaghan, P. (2007) " Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109 - 120, June 2007, Prague.
- [6] Frost, R., Hafiz, R. and Callaghan, P. (2008) " Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902/2008, Pages: 167 - 181, January 2008, San Francisco.
- [7] <http://www.cs.uwindsor.ca/~hafiz/proHome.html>

Further reading

- Chapman, Nigel P., *LR Parsing: Theory and Practice* (<http://books.google.com/books?id=nEA9AAAAIAAJ&printsec=frontcover>), Cambridge University Press, 1987. ISBN 0-521-30413-X
- Grune, Dick; Jacobs, Ceriel J.H., *Parsing Techniques - A Practical Guide* (http://dickgrune.com/Books/PTAPG_1st_Edition/), Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. Originally published by Ellis Horwood, Chichester, England, 1990; ISBN 0-13-651431-6

External links

- Objective-C parser written in C# (<http://code.google.com/p/nobjectiveast/>)
- The Lemon LALR Parser Generator (<http://www.hwaci.com/sw/lemon/>)

Lookahead

For information about the look-ahead feature of some audio compressors, see look-ahead.

Lookahead is a tool in algorithms for looking ahead a few more input items before making a cost effective decision at one stage of the algorithm.

Applications

Lookahead in search problems

In artificial intelligence, **lookahead** is an important component of combinatorial search which specifies, roughly, how deeply the graph representing the problem is explored. The need for a specific limit on lookahead comes from the large problem graphs in many applications, such as computer chess and computer Go. A naive breadth-first search of these graphs would quickly consume all the memory of any modern computer. By setting a specific lookahead limit, the algorithm's time can be carefully controlled; its time increases exponentially as the lookahead limit increases.

More sophisticated search techniques such as alpha-beta pruning are able to eliminate entire subtrees of the search tree from consideration. When these techniques are used, lookahead is not a precisely defined quantity, but instead either the maximum depth searched or some type of average.

Lookahead in parsing

Lookahead establishes the maximum incoming tokens that a parser can use to decide which rule it should use.

Lookahead is especially relevant to LL, LR, and LALR parsers, where it is often explicitly indicated by affixing the lookahead to the algorithm name in parentheses, such as LALR(1).

Most programming languages, the primary target of parsers, are carefully defined in such a way that a parser with limited lookahead, typically one, can parse them, because parsers with limited lookahead are often more efficient. One important change to this trend came in 1990 when Terence Parr created ANTLR for his Ph.D. thesis, a parser generator for efficient LL(k) parsers, where k is any fixed value.

Parsers typically have only a few actions after seeing each token. They are shift (add this token to the stack for later reduction), reduce (pop tokens from the stack and form a syntactic construct), end, error (no known rule applies) or conflict (does not know whether to shift or reduce).

Lookahead has two advantages.

- It helps the parser take the correct action in case of conflicts. For example, parsing the if statement in the case of an else clause.
- It eliminates many duplicate states and eases the burden of an extra stack. A C language non-lookahead parser will have around 10,000 states. A lookahead parser will have around 300 states.

Example: Parsing the Expression $1 + 2 * 3$

```
Set of expression parsing rules (called grammar) is as follows,
```

Rule1:	$E \rightarrow E + E$	Expression is the sum of two expressions.
Rule2:	$E \rightarrow E * E$	Expression is the product of two expressions.
Rule3:	$E \rightarrow \text{number}$	Expression is a simple number
Rule4:	+ has less precedence than *	

Most programming languages (except for a few such as APL and Smalltalk) and algebraic formulas give higher precedence to multiplication than addition, in which case the correct interpretation of the example above is $(1 + (2*3))$. Note that Rule4 above is a semantic rule. It is possible to rewrite the grammar to incorporate this into the syntax. However, not all such rules can be translated into syntax.

Simple non-lookahead parser actions

1. Reduces 1 to expression E on input 1 based on rule3.
2. Shift + onto stack on input 1 in anticipation of rule1.
3. Reduce stack element 2 to Expression E based on rule3.
4. Reduce stack items E+ and new input E to E based on rule1.
5. Shift * onto stack on input * in anticipation of rule2.
6. Shift 3 onto stack on input 3 in anticipation of rule3.
7. Reduce 3 to Expression E on input 3 based on rule3.
8. Reduce stack items E* and new input E to E based on rule2.

The parse tree and resulting code from it is not correct according to language semantics.

To correctly parse without lookahead, there are three solutions:

- The user has to enclose expressions within parentheses. This often is not a viable solution.
- The parser needs to have more logic to backtrack and retry whenever a rule is violated or not complete. The similar method is followed in LL parsers.
- Alternatively, the parser or grammar needs to have extra logic to delay reduction and reduce only when it is absolutely sure which rule to reduce first. This method is used in LR parsers. This correctly parses the expression but with many more states and increased stack depth.

Lookahead parser actions

1. Shift 1 onto stack on input 1 in anticipation of rule3. It does not reduce immediately.
2. Reduce stack item 1 to simple Expression on input + based on rule3. The lookahead is +, so we are on path to E +, so we can reduce the stack to E.
3. Shift + onto stack on input + in anticipation of rule1.
4. Shift 2 onto stack on input 2 in anticipation of rule3.
5. Reduce stack item 2 to Expression on input * based on rule3. The lookahead * expects only E before it.
6. Now stack has E + E and still the input is *. It has two choices now, either to shift based on rule2 or reduction based on rule1. Since * has more precedence than + based on rule4, so shift * onto stack in anticipation of rule2.
7. Shift 3 onto stack on input 3 in anticipation of rule3.
8. Reduce stack item 3 to Expression after seeing end of input based on rule3.
9. Reduce stack items E * E to E based on rule2.
10. Reduce stack items E + E to E based on rule1.

The parse tree generated is correct and simply more efficient than non-lookahead parsers. This is the strategy followed in LALR parsers.

Lookahead vs. Lazy evaluation

This is in contrast to another technique called lazy evaluation that delays the computation until it is really needed. Both techniques are used for economical usage of space or time. Lookahead makes the right decision and so avoids backtracking from undesirable stages at later stages of the algorithm and so saves space, at the cost of a slight increase of time due to the overhead of extra lookups. Lazy evaluation normally skips the unexplored algorithmic paths and thus saves both the time and space in general. Some applications of lazy evaluations are demand paging in operating systems and lazy parse tables in compilers.

In search space exploration, both the techniques are used. When there are already promising paths in the algorithm to evaluate, lazy evaluation is used and to be explored paths will be saved in the queue or stack. When there are no promising paths to evaluate and to check whether the new path can be a more promising path in leading to solution, lookahead is used.

Compilers also use both the techniques. They will be lazy in generating parse tables from given rules, but they lookahead in parsing given input.

Symbol table

In computer science, a **symbol table** is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.

Implementation

A common implementation technique is to use a hash table implementation. A compiler may use one large symbol table for all symbols or use separated, hierarchical symbol tables for different scopes.

Uses

An object file will contain a symbol table of the identifiers it contains that are externally visible. During the linking of different object files, a linker will use these symbol tables to resolve any unresolved references.

A symbol table may only exist during the translation process, or it may be embedded in the output of that process for later exploitation, for example, during an interactive debugging session, or as a resource for formatting a diagnostic report during or after execution of a program.

While reverse engineering an executable, many tools refer to the symbol table to check what addresses have been assigned to global variables and known functions. If the symbol table has been stripped or cleaned out before being converted into an executable, tools will find it harder to determine addresses or understand anything about the program.

At the time of accessing variables and allocating memory dynamically, a compiler should perform many works and as such the extended stack model requires the **symbol table**.

Example

The symbol table of a small program is listed below. The table itself was generated using the GNU binutils' nm utility. There is one data symbol, holaamigosh (noted by the "D" type), and many functions (self defined as well as from the standard library). The first column is where the symbol is located in the memory, the second is "The symbol type^[1]" and the third is the name of the symbol. By passing suitable parameters, the symbol table was made to sort on basis of address.

Example table

Address	Type	Name
00000020	a	T_BIT
00000040	a	F_BIT
00000080	a	I_BIT
20000004	t	irqvec
20000008	t	fiqvec
2000000c	t	InitReset
20000018	T	_main
20000024	t	End
20000030	T	AT91F_US3_CfgPIO_useB
2000005c	t	AT91F_PIO_CfgPeriph

200000b0	T	main
20000120	T	AT91F_DBGU_Printf
20000190	t	AT91F_US_TxReady
200001e0	t	AT91F_US_PutChar
200001f8	T	AT91F_SpuriousHandler
20000214	T	AT91F_DataAbort
20000230	T	AT91F_FetchAbort
2000024c	T	AT91F_Undef
20000268	T	AT91F_UndefHandler
20000284	T	AT91F_LowLevelInit
200002e0	t	AT91F_DBGU_CfgPIO
2000030c	t	AT91F_PIO_CfgPeriph
20000360	t	AT91F_US_Configure
200003dc	t	AT91F_US_SetBaudrate
2000041c	t	AT91F_US_Baudrate
200004ec	t	AT91F_US_SetTimeguard
2000051c	t	AT91F_PDC_Open
2000059c	t	AT91F_PDC_DisableRx
200005c8	t	AT91F_PDC_DisableTx
200005f4	t	AT91F_PDC_SetNextTx
20000638	t	AT91F_PDC_SetNextRx
2000067c	t	AT91F_PDC_SetTx
200006c0	t	AT91F_PDC_SetRx
20000704	t	AT91F_PDC_EnableRx
20000730	t	AT91F_PDC_EnableTx
2000075c	t	AT91F_US_EnableTx
20000788	T	__aeabi_uidiv
20000788	T	__udivsi3
20000884	T	__aeabi_uidivmod
2000089c	T	__aeabi_idiv0
2000089c	T	__aeabi_ldiv0
2000089c	T	__div0
200009a0	D	_data
200009a0	A	_etext
200009a0	D	holaamigosh
200009a4	A	__bss_end__
200009a4	A	__bss_start
200009a4	A	__bss_start__
200009a4	A	_edata

200009a4	A	_end
----------	---	------

References

[1] <http://sourceware.org/binutils/docs-2.17/binutils/nm.html#nm>

Abstract syntax

The **abstract syntax** of data is its structure described as a data type (possibly, but not necessarily, an abstract data type), independent of any particular representation or encoding.

To be implemented either for computation or communications, a mapping from the abstract syntax to specific machine representations and encodings must be defined; these may be called the "concrete syntax" (in language implementation) or the "transfer syntax" (in communications).

A compiler's internal representation of a program will typically be specified by an abstract syntax in terms of categories such as "statement", "expression" and "identifier". This is independent of the source syntax (**concrete syntax**) of the language being compiled (though it will often be very similar). A parse tree is similar to an abstract syntax tree but it will typically also contain features such as parentheses which are syntactically significant but which are implicit in the structure of the abstract syntax tree.

Algebraic data types are particularly well-suited to the implementation of abstract syntax.

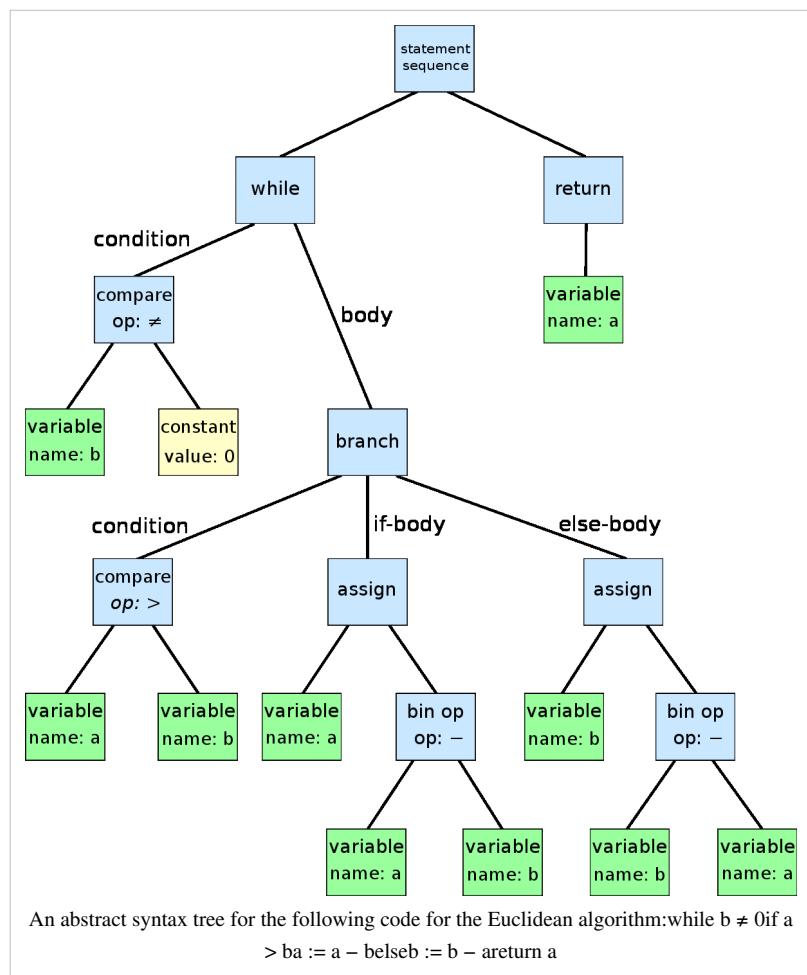
References

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

Abstract syntax tree

In computer science, an **abstract syntax tree** (AST), or just **syntax tree**, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is 'abstract' in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct such as an if-condition-then expression may be denoted by a single node with two branches.

This makes abstract syntax trees different from concrete syntax trees, traditionally called parse trees, which are often built by a parser as part of the source code translation and compiling process. Once built, additional information is added to the AST by subsequent processing, e.g., contextual analysis.



References

- This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

External links

- AST View ^[1], an Eclipse plugin to visualize a Java abstract syntax tree
- Good information about the Eclipse AST and Java Code Manipulation ^[2]
- Paper "Abstract Syntax Tree Implementation Idioms" ^[3] by Joel Jones (overview of AST implementation in various language families)
- Paper "Abstract Syntax Tree Design" ^[4] by Nicola Howarth (note that this merely presents the design of one particular project's AST, and is not generally informative)
- Paper "Understanding source code evolution using abstract syntax tree matching" ^[5] by Iulian Neamtiu, Jeffrey S. Foster and Michael Hicks
- Paper "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction" ^[6] by Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall.
- Diploma thesis "Improving Abstract Syntax Tree based Source Code Change Detection" ^[7] by Michael Würsch

- Article "Thoughts on the Visual C++ Abstract Syntax Tree (AST)"^[8] by Jason Lucas
- Tutorial "Abstract Syntax Tree Metamodel Standard"^[9]
- PMD^[10] uses AST representation to control code source quality
- CAST representation^[11]
- Abstract Syntax Tree Unparsing^[12]

References

- [1] <http://www.eclipse.org/jdt/ui/astview/index.php>
- [2] http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html
- [3] <http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Jones-ImplementingASTs.pdf>
- [4] <http://www.ansa.co.uk/ANSATech/95/Primary/155101.pdf>
- [5] <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.5815&rep=rep1&type=pdf>
- [6] http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Publications/fluri-changedistilling.pdf
- [7] <http://seal.ifi.unizh.ch/137/>
- [8] <http://blogs.msdn.com/vcblog/archive/2006/08/16/702823.aspx>
- [9] http://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf
- [10] <http://pmd.sourceforge.net>
- [11] <http://www.cs.utah.edu/flux/flick/current/doc/guts/gutsch6.html>
- [12] http://eli-project.sourceforge.net/elionline/idem_3.html

Context-free grammar

In formal language theory, a **context-free grammar (CFG)** is a formal grammar in which every production rule is of the form

$$V \rightarrow w$$

where V is a single nonterminal symbol, and w is a string of terminals and/or nonterminals (w can be empty).

The languages generated by context-free grammars are known as the context-free languages.

Context-free grammars are important in linguistics for describing the structure of sentences and words in natural language, and in computer science for describing the structure of programming languages and other artificial languages.

In linguistics, some authors use the term **phrase structure grammar** to refer to context-free grammars. In computer science, a popular notation for context-free grammars is Backus–Naur Form, or *BNF*.

Background

Since the time of Pāṇini, at least, linguists have described the grammars of languages in terms of their block structure, and described how sentences are recursively built up from smaller phrases, and eventually individual words or word elements.

An essential property of these block structures is that logical units never overlap. For example, the sentence:

John, whose blue car was in the garage, walked to the green store.

can be logically parenthesized as follows:

(John, ((whose blue car) (was (in the garage)))), (walked (to (the green store))).

A context-free grammar provides a simple and mathematically precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the "block structure" of sentences in a natural way. Its simplicity makes the formalism amenable to rigorous mathematical study. Important features of natural language syntax such as agreement and reference are not part of the context-free grammar, but the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of

adjectives and adverbs are swallowed by nouns and verbs, is described exactly.

The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky, and also their classification as a special type of formal grammar (which he called phrase-structure grammars). [1]

In Chomsky's generative grammar framework, the syntax of natural language was described by a context-free rules combined with transformation rules. In later work (e.g. Chomsky 1981), the idea of formulating a grammar consisting of explicit rewrite rules was abandoned. In other generative frameworks, e.g. Generalized Phrase Structure Grammar (Gazdar et al. 1985), context-free grammars were taken to be the mechanism for the entire syntax, eliminating transformations.

Block structure was introduced into computer programming languages by the Algol project (1957-1960), which, as a consequence, also featured a context-free grammar to describe the resulting Algol syntax. This became a standard feature of computer languages, and the notation for grammars used in concrete descriptions of computer languages came to be known as Backus-Naur Form, after two members of the Algol language design committee.

The "block structure" aspect that context-free grammars capture is so fundamental to grammar that the terms syntax and grammar are often identified with context-free grammar rules, especially in computer science. Formal constraints not captured by the grammar are then considered to be part of the "semantics" of the language.

Context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar. An Earley parser is an example of such an algorithm, while the widely used LR and LL parsers are simpler algorithms that deal only with more restrictive subsets of context-free grammars.

Formal definitions

A context-free grammar G is defined by the 4-tuple:

$$G = (V, \Sigma, R, S) \text{ where}$$

1. V is a finite set; each element $v \in V$ is called a *non-terminal character* or a *variable*. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G .
2. Σ is a finite set of *terminals*, disjoint from V , which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G .
3. R is a finite relation from V to $(V \cup \Sigma)^*$. The members of R are called the (*rewrite*) *rules* or *productions* of the grammar.
4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V .

The asterisk represents the Kleene star operation.

Rule application

For any strings $u, v \in (V \cup \Sigma)^*$, we say u yields v , written as $u \Rightarrow v$, if $\exists(\alpha, \beta) \in R$ and $u_1, u_2 \in (V \cup \Sigma)^*$ such that $u = u_1\alpha u_2$ and $v = u_1\beta u_2$. Thus, v is the result of applying the rule (α, β) to u .

Repetitive rule application

For any $u, v \in (V \cup \Sigma)^*$, $u \xrightarrow{*} v$ (or $u \Rightarrow \Rightarrow v$ in some textbooks), if $\exists u_1, u_2, \dots, u_k \in (V \cup \Sigma)^*$, $k \geq 0$ such that $u \Rightarrow u_1 \Rightarrow u_2 \dots \Rightarrow u_k \Rightarrow v$

Context-free language

The language of a grammar $G = (V, \Sigma, R, S)$ is the set

$$L(G) = \{w \in \Sigma^* : S \xrightarrow{*} w\}$$

A language L is said to be a context-free language (CFL), if there exists a CFG G , such that $L = L(G)$.

Proper CFGs

A context-free grammar is said to be *proper*, if it has

- no *inaccessible* symbols: $\forall N \in V : \exists \alpha, \beta \in V^* : S \xrightarrow{*} \alpha N \beta$
- no *unproductive* symbols: $\forall N \in V : \exists w \in \Sigma^* : N \xrightarrow{*} w$
- no ϵ -productions: $\forall N \in V, w \in \Sigma^* : (N, w) \in R \Rightarrow w \neq \epsilon$
- no cycles: $\neg \exists N \in V : N \xrightarrow{*} N$

The grammar $G = (\{S\}, \{a, b\}, S, P)$, with productions

$$S \rightarrow aSa,$$

$$S \rightarrow bSb,$$

$$S \rightarrow \epsilon,$$

is context-free. A typical derivation in this grammar is $S \rightarrow aSa \rightarrow aaSaa \rightarrow aabSbaa \rightarrow aabbbaa$. This makes it clear that $L(G) = \{ww^R : w \in \{a, b\}^*\}$. The language is context-free, however it can be proved that it is not regular.

Examples

Well-formed parentheses

The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

The first rule allows Ss to multiply; the second rule allows Ss to become enclosed by matching parentheses; and the third rule terminates the recursion.

Starting with S, and applying the rules, one can construct:

$$S \rightarrow SS \rightarrow SSS \rightarrow (S)SS \rightarrow ((S))SS \rightarrow (((S))S(S))$$

$$\rightarrow (((S))S(S)) \rightarrow (((((S))))S(S)) \rightarrow (((((S))))((S)))$$

$$\rightarrow (((((S))))((S)))$$

Well-formed nested parentheses and square brackets

A second canonical example is two different kinds of matching nested parentheses, described by the productions:

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow () \\ S &\rightarrow (S) \\ S &\rightarrow [] \\ S &\rightarrow [S] \end{aligned}$$

with terminal symbols [] () and nonterminal S.

The following sequence can be derived in that grammar:

$$([[()] []])([])]$$

However, there is no context-free grammar for generating all sequences of two different types of parentheses, each separately balanced disregarding the other, but where the two types need not nest inside one another, for example:

$$[[[[((())])]) ([]) ([]) ([]) ([]) ([])]$$

A regular grammar

$$\begin{aligned} S &\rightarrow a \\ S &\rightarrow aS \\ S &\rightarrow bS \end{aligned}$$

The terminals here are *a* and *b*, while the only non-terminal is S. The language described is all nonempty strings of *a*s and *b*s that end in *a*.

This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side.

Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.

It is common to list all right-hand sides for the same left-hand side on the same line, using | to separate them. Hence the grammar above can be described more tersely as follows:

$$S \rightarrow a | aS | bS$$

Matching pairs

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow ab \end{aligned}$$

This grammar generates the language $\{a^n b^n : n \geq 1\}$, which is not regular (according to the Pumping Lemma for regular languages).

The special character ϵ stands for the empty string. By changing the above grammar to

$$S \rightarrow aSb | \epsilon$$

we obtain a grammar generating the language $\{a^n b^n : n \geq 0\}$ instead. This differs only in that it contains the empty string while the original grammar did not.

Algebraic expressions

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. $S \rightarrow x$
2. $S \rightarrow y$
3. $S \rightarrow z$
4. $S \rightarrow S + S$
5. $S \rightarrow S - S$
6. $S \rightarrow S * S$
7. $S \rightarrow S / S$
8. $S \rightarrow (S)$

This grammar can, for example, generate the string

$(x + y) * x - z * y / (x + x)$

as follows:

```

S (the start symbol)
→ S - S (by rule 5)
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)
→ ( S ) * S - S / S (by rule 8, applied to the leftmost S)
→ ( S ) * S - S / ( S ) (by rule 8, applied to the rightmost S)
→ ( S + S ) * S - S / ( S ) (etc.)
→ ( S + S ) * S - S * S / ( S )
→ ( S + S ) * S - S * S / ( S + S )
→ ( x + S ) * S - S * S / ( S + S )
→ ( x + y ) * S - S * S / ( S + S )
→ ( x + y ) * x - S * y / ( S + S )
→ ( x + y ) * x - S * y / ( x + S )
→ ( x + y ) * x - z * y / ( x + S )
→ ( x + y ) * x - z * y / ( x + x )

```

Note that many choices were made underway as to which rewrite was going to be performed next. These choices look quite arbitrary. As a matter of fact, they are, in the sense that the string finally generated is always the same. For example, the second and third rewrites

```

→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)

```

could be done in the opposite order:

```

→ S - S / S (by rule 7, applied to the rightmost S)
→ S * S - S / S (by rule 6, applied to the leftmost S)

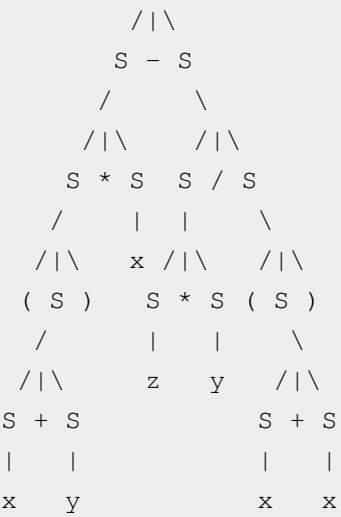
```

Also, many choices were made on which rule to apply to each selected S. Changing the choices made and not only the order they were made in usually affects which terminal string comes out at the end.

Let's look at this in more detail. Consider the parse tree of this derivation:

S

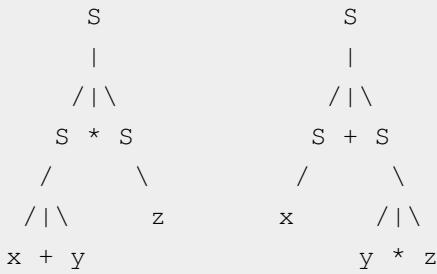
|



Starting at the top, step by step, an S in the tree is expanded, until no more unexpanded Ses (non-terminals) remain. Picking a different order of expansion will produce a different derivation, but the same parse tree. The parse tree will only change if we pick a different rule to apply at some position in the tree.

But can a different parse tree still produce the same terminal string, which is $(x + y)^* x - z^* y / (x + x)$ in this case? Yes, for this particular grammar, this is possible. Grammars with this property are called ambiguous.

For example, $x + y^* z$ can be produced with these two different parse trees:



However, the *language* described by this grammar is not inherently ambiguous: an alternative, unambiguous grammar can be given for the language, for example:

$$\begin{aligned}
T &\rightarrow x \\
T &\rightarrow y \\
T &\rightarrow z \\
S &\rightarrow S + T \\
S &\rightarrow S - T \\
S &\rightarrow S * T \\
S &\rightarrow S / T \\
T &\rightarrow (S) \\
S &\rightarrow T
\end{aligned}$$

(once again picking S as the start symbol).

Further examples

Example 1

A context-free grammar for the language consisting of all strings over {a,b} containing an unequal number of a's and b's:

$$\begin{aligned} S &\rightarrow U \mid V \\ U &\rightarrow TaU \mid TaT \\ V &\rightarrow TbV \mid TbT \\ T &\rightarrow aTbT \mid bTaT \mid \epsilon \end{aligned}$$

Here, the nonterminal T can generate all strings with the same number of a's as b's, the nonterminal U generates all strings with more a's than b's and the nonterminal V generates all strings with fewer a's than b's.

Example 2

Another example of a non-regular language is $\{b^n a^m b^{2n} : n \geq 0, m \geq 0\}$. It is context-free as it can be generated by the following context-free grammar:

$$\begin{aligned} S &\rightarrow bSbb \mid A \\ A &\rightarrow aA \mid \epsilon \end{aligned}$$

Other examples

The formation rules for the terms and formulas of formal logic fit the definition of context-free grammar, except that the set of symbols may be infinite and there may be more than one start symbol.

Context-free grammars are not limited in application to mathematical ("formal") languages. For example, it has been suggested that a class of Tamil poetry called Venpa is described by a context-free grammar.^[2]

Derivations and syntax trees

A *derivation* of a string for a grammar is a sequence of grammar rule applications, that transforms the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

A derivation is fully determined by giving, for each step:

- the rule applied in that step
- the occurrence of its right hand side to which it is applied

For clarity, the intermediate string is usually given as well.

For instance, with the grammar:

$$\begin{aligned} (1) \quad S &\rightarrow S + S \\ (2) \quad S &\rightarrow 1 \\ (3) \quad S &\rightarrow a \end{aligned}$$

the string

1 + 1 + a

can be derived with the derivation:

$$\begin{aligned} S & \\ \rightarrow & (\text{rule 1 on first } S) \\ S+S & \\ \rightarrow & (\text{rule 1 on second } S) \end{aligned}$$

```

S+S+S
→ (rule 2 on second S)
S+1+S
→ (rule 3 on third S)
S+1+a
→ (rule 2 on first S)
1+1+a

```

Often, a strategy is followed that deterministically determines the next nonterminal to rewrite:

- in a *leftmost derivation*, it is always the leftmost nonterminal;
- in a *rightmost derivation*, it is always the rightmost nonterminal.

Given such a strategy, a derivation is completely determined by the sequence of rules applied. For instance, the leftmost derivation

```

S
→ (rule 1 on first S)
S+S
→ (rule 2 on first S)
1+S
→ (rule 1 on first S)
1+S+S
→ (rule 2 on first S)
1+1+S
→ (rule 3 on first S)
1+1+a

```

can be summarized as

```
rule 1, rule 2, rule 1, rule 2, rule 3
```

The distinction between leftmost derivation and rightmost derivation is important because in most parsers the transformation of the input is defined by giving a piece of code for every grammar rule that is executed whenever the rule is applied. Therefore it is important to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the pieces of code will be executed. See for an example LL parsers and LR parsers.

A derivation also imposes in some sense a hierarchical structure on the string that is derived. For example, if the string "1 + 1 + a" is derived according to the leftmost derivation:

```

S → S + S (1)
→ 1 + S (2)
→ 1 + S + S (1)
→ 1 + 1 + S (2)
→ 1 + 1 + a (3)

```

the structure of the string would be:

$$\{ \{ 1 \}_S + \{ \{ 1 \}_S + \{ a \}_S \}_S \}_S$$

where $\{ \dots \}_S$ indicates a substring recognized as belonging to S. This hierarchy can also be seen as a tree:

```

S
/ | \

```

```

    / | \
    / | \
S  '+'  S
|      / | \
|      / | \
'1'   S  '+'  S
|      |
'1'   'a'

```

This tree is called a *concrete syntax tree* (see also abstract syntax tree) of the string. In this case the presented leftmost and the rightmost derivations define the same syntax tree; however, there is another (rightmost) derivation of the same string

$$\begin{aligned} S &\rightarrow S + S \text{ (1)} \\ &\rightarrow S + a \text{ (3)} \\ &\rightarrow S + S + a \text{ (1)} \\ &\rightarrow S + 1 + a \text{ (2)} \\ &\rightarrow 1 + 1 + a \text{ (2)} \end{aligned}$$

and this defines the following syntax tree:

```

      S
      / \
      / | \
      /   |   \
      /     |     \
S   '+'   S
      / \
      / \
      S  '+'  S  'a'
      |   |
'1'   '1'

```

If, for certain strings in the language of the grammar, there is more than one parsing tree, then the grammar is said to be an *ambiguous grammar*. Such grammars are usually hard to parse because the parser cannot always decide which grammar rule it has to apply. Usually, ambiguity is a feature of the grammar, not the language, and an unambiguous grammar can be found that generates the same context-free language. However, there are certain languages that can only be generated by ambiguous grammars; such languages are called inherently ambiguous.

Normal forms

Every context-free grammar that does not generate the empty string can be transformed into one in which no rule has the empty string as a product [a rule with ϵ as a product is called an ϵ -production]. If it does generate the empty string, it will be necessary to include the rule $S \rightarrow \epsilon$, but there need be no other ϵ -rule. Every context-free grammar with no ϵ -production has an equivalent grammar in Chomsky normal form or Greibach normal form. "Equivalent" here means that the two grammars generate the same language.

Because of the especially simple form of production rules in Chomsky Normal Form grammars, this normal form has both theoretical and practical implications. For instance, given a context-free grammar, one can use the Chomsky Normal Form to construct a polynomial-time algorithm that decides whether a given string is in the language represented by that grammar or not (the CYK algorithm).

Undecidable problems

Some questions that are undecidable for wider classes of grammars become decidable for context-free grammars; e.g. the emptiness problem (whether the grammar generates any terminal strings at all), is undecidable for context-sensitive grammars, but decidable for context-free grammars.

Still, many problems remain undecidable. Examples:

Universality

Given a CFG, does it generate the language of all strings over the alphabet of terminal symbols used in its rules?

A reduction can be demonstrated to this problem from the well-known undecidable problem of determining whether a Turing machine accepts a particular input (the Halting problem). The reduction uses the concept of a *computation history*, a string describing an entire computation of a Turing machine. We can construct a CFG that generates all strings that are not accepting computation histories for a particular Turing machine on a particular input, and thus it will accept all strings only, if the machine doesn't accept that input.

Language equality

Given two CFGs, do they generate the same language?

The undecidability of this problem is a direct consequence of the previous: we cannot even decide whether a CFG is equivalent to the trivial CFG defining the language of all strings.

Language inclusion

Given two CFGs, can the first generate all strings that the second can generate?

If this problem is decidable, then we could use it to determine whether two CFGs G1 and G2 generate the same language by checking whether L(G1) is a subset of L(G2) and L(G2) is a subset of L(G1).

Being in a lower level of the Chomsky hierarchy

Given a context-sensitive grammar, does it describe a context-free language? Given a context-free grammar, does it describe a regular language?

Each of these problems is undecidable.

Extensions

An obvious way to extend the context-free grammar formalism is to allow nonterminals to have arguments, the values of which are passed along within the rules. This allows natural language features such as agreement and reference, and programming language analogs such as the correct use and definition of identifiers, to be expressed in a natural way. E.g. we can now easily express that in English sentences, the subject and verb must agree in number.

In computer science, examples of this approach include affix grammars, attribute grammars, indexed grammars, and Van Wijngaarden two-level grammars.

Similar extensions exist in linguistics.

Another extension is to allow additional terminal symbols to appear at the left hand side of rules, constraining their application. This produces the formalism of context-sensitive grammars.

Restrictions

There are a number of important subclasses of the context-free grammars:

- LR(k) grammars (also known as deterministic context-free grammars) allow parsing (string recognition) with deterministic pushdown automata, but they can only describe deterministic context-free languages.
- Simple LR, Look-Ahead LR grammars are subclasses that allow further simplification of parsing.
- LL(k) and LL(*) grammars allow parsing by direct construction of a leftmost derivation as described above, and describe even fewer languages.
- Simple grammars are a subclass of the LL(1) grammars mostly interesting for its theoretical property that language equality of simple grammars is decidable, while language inclusion is not.
- Bracketed grammars have the property that the terminal symbols are divided into left and right bracket pairs that always match up in rules.
- Linear grammars have no rules with more than one nonterminal in the right hand side.
- Regular grammars are a subclass of the linear grammars and describe the regular languages, i.e. they correspond to finite automata and regular expressions.

LR parsing extends LL parsing to support a larger range of grammars; in turn, generalized LR parsing extends LR parsing to support arbitrary context-free grammars. On LL grammars and LR grammars, it essentially performs LL parsing and LR parsing, respectively, while on nondeterministic grammars, it is as efficient as can be expected. Although GLR parsing was developed in the 1980s, many new language definitions and parser generators continue to be based on LL, LALR or LR parsing up to the present day.

Linguistic applications

Chomsky initially hoped to overcome the limitations of context-free grammars by adding transformation rules.^[1]

Such rules are another standard device in traditional linguistics; e.g. passivization in English. Much of generative grammar has been devoted to finding ways of refining the descriptive mechanisms of phrase-structure grammar and transformation rules such that exactly the kinds of things can be expressed that natural language actually allows. Allowing arbitrary transformations doesn't meet that goal: they are much too powerful, being Turing complete unless significant restrictions are added (e.g. no transformations that introduce and then rewrite symbols in a context-free fashion).

Chomsky's general position regarding the non-context-freeness of natural language has held up since then,^[3] although his specific examples regarding the inadequacy of context-free grammars (CFGs) in terms of their weak generative capacity were later disproved.^[4] Gerald Gazdar and Geoffrey Pullum have argued that despite a few non-context-free constructions in natural language (such as cross-serial dependencies in Swiss German^[3] and reduplication in Bambara^[5]), the vast majority of forms in natural language are indeed context-free.^[4]

Notes

- [1] Chomsky, Noam (Sep 1956). "Three models for the description of language" (<http://ieeexplore.ieee.org/iel5/18/22738/01056813.pdf?isnumber=22738&prod=STD&arnumber=1056813&arnumber=1056813&arSt=+113&aref=+124&arAuthor=+Chomsky,+N.>). *Information Theory, IEEE Transactions* **2** (3): 113–124. doi:10.1109/TIT.1956.1056813. . Retrieved 2007-06-18.
- [2] L. BalaSundaraRaman; Ishwar.S, Sanjeeth Kumar Ravindranath (2003-08-22). "Context Free Grammar for Natural Language Constructs - An implementation for Venpa Class of Tamil Poetry" (<http://citeseer.ist.psu.edu/balasundararaman03context.html>). *Proceedings of Tamil Internet, Chennai, 2003*. International Forum for Information Technology in Tamil. pp. 128–136. . Retrieved 2006-08-24.
- [3] Shieber, Stuart (1985). "Evidence against the context-freeness of natural language" (<http://www.eecs.harvard.edu/~shieber/Biblio/Papers/shieber85.pdf>). *Linguistics and Philosophy* **8** (3): 333–343. doi:10.1007/BF00630917..
- [4] Pullum, Geoffrey K.; Gerald Gazdar (1982). "Natural languages and context-free languages". *Linguistics and Philosophy* **4** (4): 471–504. doi:10.1007/BF00360802.
- [5] Culy, Christopher (1985). "The Complexity of the Vocabulary of Bambara". *Linguistics and Philosophy* **8** (3): 345–351. doi:10.1007/BF00630918.

References

- Chomsky, Noam (Sept. 1956). "Three models for the description of language". *Information Theory, IEEE Transactions* **2** (3).
- Chomsky, Noam (1957), *Syntactic Structures*, Den Haag: Mouton.
- Chomsky, Noam (1965), *Aspects of the Theory of Syntax*, Cambridge (Mass.): MIT Press.
- Chomsky, Noam (1981), *Lectures on Government and Binding*, Dordrecht: Foris.
- Gazdar, Gerald; Klein, Ewan; Pullum, Geoffrey & Sag, Ivan (1985), *Generalized Phrase Structure Grammar*, Cambridge (Mass.): Harvard University Press.

Further reading

- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X.
Section 2.1: Context-Free Grammars, pp. 91–101. Section 4.1.2: Decidable problems concerning context-free languages, pp. 156–159. Section 5.1.1: Reductions via computation histories: pp. 176–183.

Terminal and nonterminal symbols

In computer science, **terminal and nonterminal symbols** are the lexical elements used in specifying the production rules that constitute a formal grammar. The terminals and nonterminals of a particular grammar are two disjoint sets.

Terminal symbols

Terminal symbols are literal characters that can appear in the inputs to or outputs from the production rules of a formal grammar and that cannot be broken down into "smaller" units. To be precise, terminal symbols cannot be changed using the rules of the grammar. For example, a grammar that is defined by two rules:

1. x can become xa
2. x can become ax

has a as a terminal symbol because no rule exists that would change it to something else. (On the other hand, x has two rules that can change it, so it is nonterminal.) A formal language defined (or *generated*) by a particular grammar is the set of strings that can be produced by the grammar *and that consist only of terminal symbols*; nonterminals that do not consist entirely of terminals may not appear in the lexemes that are said to belong to the language.

In the context of syntax analysis, as opposed to the theory of programming languages and compilers, the terms "*terminal symbol*" and "*token*" are often treated as synonymous. Quoting the so-called Dragon Book (a standard reference on the latter subject):

In a compiler, the lexical analyzer reads the characters of the source program, groups them into lexically meaningful units called lexemes, and produces as output tokens representing these lexemes. A token consists of two components, a token name and an attribute value. The token names are abstract symbols that are used by the parser for syntax analysis. Often, we shall call these token names terminals, since they appear as terminal symbols in the grammar for a programming language. The attribute value, if present, is a pointer to the symbol table that contains additional information about the token. This additional information is not part of the grammar, so in our discussion of syntax analysis, often we refer to tokens and terminals synonymously.^[1]

Terminal symbols, or just *terminals*, are the elementary symbols of the language defined by a formal grammar.

Nonterminal symbols

Nonterminal symbols, or just *nonterminals*, are the symbols which can be replaced; thus there are strings composed of some combination of terminal and nonterminal symbols. They may also be called simply *syntactic variables*. A formal grammar includes a *start symbol*, a designated member of the set of nonterminals from which all the strings in the language may be derived by successive applications of the production rules. In fact, the language defined by a grammar is precisely the set of *terminal* strings that can be so derived.

Context-free grammars are those grammars in which the left-hand side of each production rule consists of only a single nonterminal symbol. This restriction is non-trivial; not all languages can be generated by context-free grammars. Those that can are called context-free languages. These are exactly the languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

Production rules

A grammar is defined by production rules that specify which lexemes may replace which other lexemes; these rules may be used to generate strings, or to parse them. Each such rule has a *head*, or left-hand side, which consists of the string that may be replaced, and a *body*, or right-hand side, which consists of a string that may replace it. Rules are often written in the form $\langle \text{head} \rightarrow \text{body} \rangle$; e.g., the rule $z_0 \rightarrow z_1$ specifies that z_0 can be replaced by z_1 .

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s,^[2] ^[3] a grammar G consists of the following components:

- A finite set N of *nonterminal symbols*.
- A finite set Σ of *terminal symbols* that is disjoint from N .
- A finite set P of *production rules*, each rule of the form

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

where $*$ is the Kleene star operator and \cup denotes set union, so $(\Sigma \cup N)^*$ represents zero or more symbols, and N means one *nonterminal symbol*. That is, each production rule maps from one string of symbols to another, where the first string contains at least one nonterminal symbol. In the case that the body consists solely of the empty string—i.e., that it contains no symbols at all—it may be denoted with a special notation (often Λ , e or ϵ) in order to avoid confusion.

- A distinguished symbol $S \in N$ that is the *start symbol*.

A grammar is formally defined as the ordered quadruple $\langle N, \Sigma, P, S \rangle$. Such a formal grammar is often called a rewriting system or a phrase structure grammar in the literature.^[4] ^[5]

Example

For instance, the following represents an integer (which may be signed) expressed in a variant of Backus–Naur form:

```
<integer> ::= [ '-' ] <digit> {<digit>}
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

In this example, the symbols (-,0,1,2,3,4,5,6,7,8,9) are terminal symbols and $\langle \text{digit} \rangle$ and $\langle \text{integer} \rangle$ are nonterminal symbols.

Notes

[1] Aho, Lam, Sethi, & Ullman, *Compilers: Principles, Techniques, and Tools*, second edition; Pearson/Addison-Wesley, 2006. Box, p. 43.

[2] Chomsky, Noam (1956). "Three Models for the Description of Language". *IRE Transactions on Information Theory* 2 (2): 113–123. doi:10.1109/TIT.1956.1056813.

[3] Chomsky, Noam (1957). *Syntactic Structures*. The Hague: Mouton.

[4] Ginsburg, Seymour (1975). *Algebraic and automata theoretic properties of formal languages*. North-Holland. pp. 8–9. ISBN 0720425069.

[5] Harrison, Michael A. (1978). *Introduction to Formal Language Theory*. Reading, Mass.: Addison-Wesley Publishing Company. pp. 13. ISBN 0201029553.

References

- Aho, Lam, Sethi, & Ullman, *Compilers: Principles, Techniques, and Tools*, second edition; Pearson/Addison-Wesley, 2006. Section 2.2 (beginning on p. 42). Note that this section only discusses context-free grammars.
- http://osr507doc.sco.com/en/tools/Yacc_specs_symbols.html

Left recursion

In computer science, **left recursion** is a special case of recursion.

In terms of context-free grammar, a non-terminal x is left-recursive if the left-most symbol in any of x 's ‘alternatives’ either immediately (direct left-recursive) or through some other non-terminal definitions (indirect/hidden left-recursive) rewrites to x again.

Definition

"A grammar is left-recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-symbol."^[1]

Immediate left recursion

Immediate left recursion occurs in rules of the form

$$A \rightarrow A\alpha \mid \beta$$

where α and β are sequences of nonterminals and terminals, and β doesn't start with A . For example, the rule

$$\text{Expr} \rightarrow \text{Expr} + \text{Term}$$

is immediately left-recursive. The *recursive descent parser* for this rule might look like:

```
function Expr()
{
    Expr();  match('+');  Term();
}
```

and a recursive descent parser would fall into infinite recursion when trying to parse a grammar which contains this rule.

Indirect left recursion

Indirect left recursion in its simplest form could be defined as:

$$A \rightarrow B\alpha \mid C$$

$$B \rightarrow A\beta \mid D,$$

possibly giving the derivation $A \Rightarrow B\alpha \Rightarrow A\beta\alpha \Rightarrow \dots$

More generally, for the nonterminals A_0, A_1, \dots, A_n , indirect left recursion can be defined as being of the form:

$$A_0 \rightarrow A_1\alpha_1 \mid \dots$$

$$A_1 \rightarrow A_2\alpha_2 \mid \dots$$

...

$$A_n \rightarrow A_0\alpha_{n+1} \mid \dots$$

where $\alpha_1, \alpha_2, \dots, \alpha_n$ are sequences of nonterminals and terminals.

Accommodating left recursion in top-down parsing

A formal grammar that contains left recursion cannot be parsed by a LL(k)-parser or other naive recursive descent parser unless it is converted to a weakly equivalent right-recursive form. In contrast, left recursion is preferred for LALR parsers because it results in lower stack usage than right recursion. However, more sophisticated top-down parsers can implement general context-free grammars by use of curtailment. In 2006, Frost and Hafiz describe an algorithm which accommodates ambiguous grammars with direct left-recursive production rules.^[2] That algorithm was extended to a complete parsing algorithm to accommodate indirect as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially-exponential number of parse trees for highly-ambiguous grammars by Frost, Hafiz and Callaghan in 2007.^[3] The authors then implemented the algorithm as a set of parser combinators written in the Haskell programming language.^[4]

Removing left recursion

Removing immediate left recursion

The general algorithm to remove immediate left recursion follows. Several improvements to this method have been made, including the ones described in "Removing Left Recursion from Context-Free Grammars", written by Robert C. Moore.^[5] For each rule of the form

$$A \rightarrow A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_m$$

where:

- A is a left-recursive nonterminal
- α is a sequence of nonterminals and terminals that is not null ($\alpha \neq \epsilon$)
- β is a sequence of nonterminals and terminals that does not start with A.

replace the A-production by the production:

$$A \rightarrow \beta_1 A' | \dots | \beta_m A'$$

And create a new nonterminal

$$A' \rightarrow \epsilon | \alpha_1 A' | \dots | \alpha_n A'$$

This newly created symbol is often called the "tail", or the "rest".

As an example, consider the rule

$$Expr \rightarrow Expr + Expr | Int | String$$

This could be rewritten to avoid left recursion as

$$Expr \rightarrow Int ExprRest | String ExprRest$$

$$ExprRest \rightarrow \epsilon | + Expr ExprRest$$

The last rule happens to be equivalent to the slightly shorter form

$$ExprRest \rightarrow \epsilon | + Expr$$

Removing indirect left recursion

If the grammar has no ϵ -productions (no productions of the form $A \rightarrow \dots | \epsilon | \dots$) and is not cyclic (no derivations of the form $A \Rightarrow \dots \Rightarrow A$ for any nonterminal A), this general algorithm may be applied to remove indirect left recursion :

Arrange the nonterminals in some (any) fixed order A_1, \dots, A_n .

```

for i = 1 to n {
    for j = 1 to i - 1 {
        • let the current  $A_j$  productions be
            
$$A_j \rightarrow \delta_1 | \dots | \delta_k$$

        • replace each production  $A_i \rightarrow A_j \gamma$  by
            
$$A_i \rightarrow \delta_1 \gamma | \dots | \delta_k \gamma$$

    }
    • remove direct left recursion for  $A_i$ 
}

```

Pitfalls

The above transformations remove left-recursion by creating a right-recursive grammar; but this changes the associativity of our rules. Left recursion makes left associativity; right recursion makes right associativity. Example : We start out with a grammar :

$$Expr \rightarrow Expr + Term | Term$$

$$Term \rightarrow Term * Factor | Factor$$

$$Factor \rightarrow (Expr) | Int$$

After having applied standard transformations to remove left-recursion, we have the following grammar :

$$Expr \rightarrow Term Expr'$$

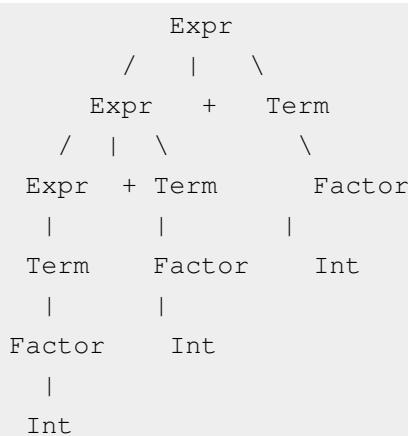
$$Expr' \rightarrow + Term Expr' | \epsilon$$

$$Term \rightarrow Factor Term'$$

$$Term' \rightarrow * Factor Term' | \epsilon$$

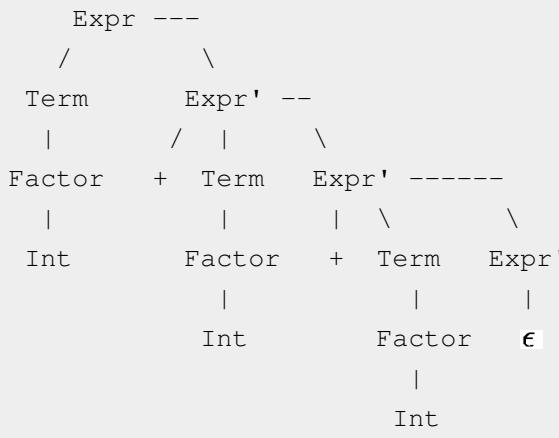
$$Factor \rightarrow (Expr) | Int$$

Parsing the string 'a + a + a' with the first grammar in an LALR parser (which can recognize left-recursive grammars) would have resulted in the parse tree:



This parse tree grows to the left, indicating that the '+' operator is left associative, representing $(a + a) + a$.

But now that we've changed the grammar, our parse tree looks like this :



We can see that the tree grows to the right, representing $a + (a + a)$. We have changed the associativity of our operator '+', it is now right-associative. While this isn't a problem for the associativity of addition, it would have a significantly different value if this were subtraction.

The problem is that normal arithmetic requires left associativity. Several solutions are: (a) rewrite the grammar to be left recursive, or (b) rewrite the grammar with more nonterminals to force the correct precedence/associativity, or (c) if using YACC or Bison, there are *operator declarations*, %left, %right and %nonassoc, which tell the parser generator which associativity to force.

References

- [1] Notes on Formal Language Theory and Parsing (<http://www.cs.may.ie/~jpower/Courses/parsing/parsing.pdf#search='indirect left recursion'>), James Power, Department of Computer Science National University of Ireland, Maynooth Maynooth, Co. Kildare, Ireland.JPR02
- [2] Frost, R.; R. Hafiz (2006). "A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time." (<http://portal.acm.org/citation.cfm?id=1149988>). *ACM SIGPLAN Notices* **41** (5): 46–54. doi:10.1145/1149982.1149988..
- [3] Frost, R.; R. Hafiz and P. Callaghan (June 2007). "Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars." (<http://acl.ldc.upenn.edu/W/W07/W07-2215.pdf>). *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* (Prague): 109–120. .
- [4] Frost, R.; R. Hafiz and P. Callaghan (January 2008). "Parser Combinators for Ambiguous Left-Recursive Grammars." (http://cs.uwindsor.ca/~richard/PUBLICATIONS/PADL_08.pdf). *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* **4902** (2008): 167–181. doi:10.1007/978-3-540-77442-6_12. .
- [5] Moore, Robert C. (May 2000). "Removing Left Recursion from Context-Free Grammars" (<http://acl.ldc.upenn.edu/A/A00/A00-2033.pdf>). *6th Applied Natural Language Processing Conference*: 249–255. .

External links

- CMU lecture on left-recursion (<http://www.cs.umd.edu/class/fall2002/cmsc430/lec4.pdf>)
- Practical Considerations for LALR(1) Grammars (<http://lambda.uta.edu/cse5317/notes/node21.html>)
- X-SAIGA (<http://www.cs.uwindsor.ca/~hafiz/proHome.html>) - eXecutable SpecificAtIons of GrAmmars

Backus–Naur Form

In computer science, **BNF (Backus Normal Form or Backus–Naur Form)** is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols. It is applied wherever exact descriptions of languages are needed, for instance, in official language specifications, in manuals, and in textbooks on programming language theory.

Many extensions and variants of the original notation are used; some are exactly defined, including *Extended Backus–Naur Form* (EBNF) and *Augmented Backus–Naur Form* (ABNF).

History

The idea of describing the structure of language with rewriting rules can be traced back to at least the work of Pāṇini (about the 4th century BC), who used it in his description of Sanskrit word structure. American linguists such as Leonard Bloomfield and Zellig Harris took this idea a step further by attempting to formalize language and its study in terms of formal definitions and procedures (around 1920–1960).

Meanwhile, string rewriting rules as formal, abstract systems were introduced and studied by mathematicians such as Axel Thue (in 1914), Emil Post (1920s–1940s) and Alan Turing (1936). Noam Chomsky, teaching linguistics to students of information theory at MIT, combined linguistics and mathematics, by taking what is essentially Thue's formalism as the basis for the description of the syntax of natural language; he also introduced a clear distinction between generative rules (those of context-free grammars) and transformation rules (1956).^[1] ^[2]

John Backus, a programming language designer at IBM, proposed "metalinguistic formulas"^[3] ^[4] to describe the syntax of the new programming language IAL, known today as ALGOL 58 (1959), using the BNF notation.

Further development of ALGOL led to ALGOL 60; in its report (1963), Peter Naur named Backus's notation **Backus Normal Form**, and simplified it to minimize the character set used. However, Donald Knuth argued that BNF should rather be read as **Backus–Naur Form**, as it is "not a normal form in any sense",^[5] unlike, for instance, Chomsky Normal Form.

Introduction

A BNF specification is a set of derivation rules, written as

```
<symbol> ::= __expression__
```

where *<symbol>* is a *nonterminal*, and the *__expression__* consists of one or more sequences of symbols; more sequences are separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.

Example

As an example, consider this possible BNF for a U.S. postal address:

```
<postal-address> ::= <name-part> <street-address> <zip-part>

<name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
               | <personal-part> <name-part>

<personal-part> ::= <first-name> | <initial> ". "
```

```

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""

```

This translates into English as:

- A postal address consists of a name-part, followed by a street-address part, followed by a zip-code part.
- A name-part consists of either: a personal-part followed by a last name followed by an optional suffix (Jr., Sr., or dynastic number) and end-of-line, or a personal part followed by a name part (this rule illustrates the use of recursion in BNFs, covering the case of people who use multiple first and middle names and/or initials).
- A personal-part consists of either a first name or an initial followed by a dot.
- A street address consists of a house number, followed by a street name, followed by an optional apartment specifier, followed by an end-of-line.
- A zip-part consists of a town-name, followed by a comma, followed by a state code, followed by a ZIP-code followed by an end-of-line.
- A opt-suffix-part consists of a suffix, such as "Sr.", "Jr." or a roman-numeral, or an empty string (i.e. nothing).

Note that many things (such as the format of a first-name, apartment specifier, ZIP-code, and Roman numeral) are left unspecified here. If necessary, they may be described using additional BNF rules.

Further examples

BNF's syntax itself may be represented with a BNF like the following:

```

<syntax> ::= <rule> | <rule> <syntax>

<rule>   ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace> "::="
            <opt-whitespace> <expression> <line-end>

<opt-whitespace> ::= " " <opt-whitespace> | "" <!-- "" is empty string, i.e. no whitespace -->

<expression>    ::= <list> | <list> "|" <expression>

<line-end>      ::= <opt-whitespace> <EOL> | <line-end> <line-end>

<list>       ::= <term> | <term> <opt-whitespace> <list>

<term>       ::= <literal> | "<" <rule-name> ">"

<literal> ::= ' ' <text> ' ' | " " <text> " " <!-- actually, the original BNF did not use quotes -->

```

This assumes that no whitespace is necessary for proper interpretation of the rule. <EOL> represents the appropriate line-end specifier (in ASCII, carriage-return and/or line-feed, depending on the operating system). <rule-name> and <text> are to be substituted with a declared rule's name/label or literal text, respectively.

In the U.S. postal address example above, the entire block-quote is a syntax. Each line or unbroken grouping of lines is a rule; for example one rule begins with "<name-part> ::=". The other part of that rule (aside from a line-end) is an expression, which consists of two lists separated by a pipe "|". These two lists consists of some terms (three terms and two terms, respectively). Each term in this particular rule is a rule-name.

Variants

There are many variants and extensions of BNF, generally either for the sake of simplicity and succinctness, or to adapt it to a specific application. One common feature of many variants is the use of regular expression repetition operators such as * and +. The Extended Backus–Naur Form (EBNF) is a common one. In fact the example above is not the pure form invented for the ALGOL 60 report. The bracket notation "[]" was introduced a few years later in IBM's PL/I definition but is now universally recognised. ABNF and RBNF are other extensions commonly used to describe Internet Engineering Task Force (IETF) protocols.

Parsing expression grammars build on the BNF and regular expression notations to form an alternative class of formal grammar, which is essentially analytic rather than generative in character.

Many BNF specifications found online today are intended to be human readable and are non-formal. These often include many of the following syntax rules and extensions:

- Optional items enclosed in square brackets. E.g. [<item-x>].
- Items repeating 0 or more times are enclosed in curly brackets or suffixed with an asterisk (*), such as "<word> ::= <letter> {<letter>}" or "<word> ::= <letter>*", respectively.
- Items repeating 1 or more times are suffixed with an addition (plus) symbol (+).
- Terminals may appear in bold and NonTerminals in plain text rather than using italics and angle brackets.
- Alternative choices in a production are separated by the 'l' symbol. E.g., <alternative-A> | <alternative-B>.
- Where items need to be grouped they are enclosed in simple parentheses.

References

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

- [1] Chomsky, Noam (1956). "Three models for the description of language" (<http://www.chomsky.info/articles/195609--.pdf>). *IRE Transactions on Information Theory* **2** (2): 113–124. doi:10.1109/TIT.1956.1056813..
- [2] Chomsky, Noam (1957). *Syntactic Structures*. The Hague: Mouton.
- [3] Backus, J.W. (1959). "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference" (http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_ProposedIAL.pdf/view). *Proceedings of the International Conference on Information Processing*. UNESCO. pp. 125–132.
- [4] Farrell, James A. (August 1995). "Extended Backus Naur Form" (<http://www.cs.man.ac.uk/~pjf/farrell/comp2.html#EBNF>). . Retrieved May 11, 2011.
- [5] Knuth, Donald E. (1964). "Backus Normal Form vs. Backus Naur Form". *Communications of the ACM* **7** (12): 735–736. doi:10.1145/355588.365140.

External links

- Backus Normal Form vs. Backus-Naur Form (<http://compilers.iecc.com/comparch/article/93-07-017>) explains some of the history of the two names.
- BNF and EBNF: What are they and how do they work? (<http://www.garshol.priv.no/download/text/bnf.html>) by Lars Marius Garshol.
- RFC 4234 Augmented BNF for Syntax Specifications: ABNF.
- RFC 5511 Routing BNF: A Syntax Used in Various Protocol Specifications.
- Comparison of different variants of BNF (<http://www-cgi.uni-regensburg.de/~brf09510/grammatypes.html>)
- Syntax diagram of EBNF (<http://www-cgi.uni-regensburg.de/~brf09510/syntax/lazyebnf.ebnf.html>)
- Generation of syntax diagrams from EBNF (<http://www-cgi.uni-regensburg.de/~brf09510/syntax.html>)
- ISO/IEC 14977:1996(E) *Information technology - Syntactic metalanguage - Extended BNF*, available from ISO (<http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html>) or from Marcus Kuhn (<http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>) (the latter is missing the cover page, but is otherwise much cleaner)

Language grammars

- Algol-60 BNF (<http://www.lrz-muenchen.de/~bernhard/Algol-BNF.html>), the original BNF.
- BNF grammars for SQL-92, SQL-99 and SQL-2003 (<http://savage.net.au/SQL/index.html>), Freely available BNF grammars for SQL.
- BNF Web Club (<http://cui.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html>), Freely available BNF grammars for SQL, Ada, Java.
- BNF Examples and Full Programming Languages (<http://www.devincook.com/GOLDParser/grammars/index.htm>), Freely available BNF grammars for programming languages (C, Java, JavaScript, C#, VB.NET, SQL-89), academic languages, file formats (HTML, XML, CSS) and others (regular expressions, YACC). These are written in the GOLD Meta-Language (<http://www.devincook.com/GOLDParser/doc/meta-language/index.htm>), consisting of BNF and Regular Expressions.
- Free Programming Language Grammars for Compiler Construction (<http://www.thefreecountry.com/sourcecode/grammars.shtml>), Freely available BNF/EBNF grammars for C/C++, Pascal, COBOL, Ada 95, PL/I.
- BNF files related to the STEP standard (<http://exp-engine.svn.sourceforge.net/viewvc/exp-engine/engine/trunk/docs/>). Includes Parts 11, 14, and 21 of the ISO 10303 (STEP) standard

Extended Backus–Naur Form

In computer science, **Extended Backus–Naur Form (EBNF)** is a family of metasyntax notations used for expressing context-free grammars: that is, a formal way to describe computer programming languages and other formal languages. They are extensions of the basic Backus–Naur Form (BNF) metasyntax notation.

The earliest EBNF was originally developed by Niklaus Wirth. However, many variants of EBNF are in use. The International Organization for Standardization has adopted an EBNF standard (ISO/IEC 14977^[1]). This article uses EBNF as specified by the ISO for examples applying to all EBNF:s. Other EBNF variants use somewhat different syntactic conventions.

Basics

EBNF is a code that expresses the grammar of a computer language. An EBNF consists of terminal symbol and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence. Examples of terminal symbols include alphanumeric characters, punctuation marks, and white space characters.

The EBNF defines production rules where sequences of symbols are respectively assigned to a nonterminal:

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
digit           = "0" | digit excluding zero ;
```

This production rule defines the nonterminal *digit* which is on the left side of the assignment. The vertical bar represents an alternative and the terminal symbols are enclosed with quotation marks followed by a semicolon as terminating character. Hence a *Digit* is a 0 or a *digit excluding zero* that can be 1 or 2 or 3 and so forth until 9.

A production rule can also include a sequence of terminals or nonterminals, each separated by a comma:

```
twelve          = "1" , "2" ;
two hundred one = "2" , "0" , "1" ;
three hundred twelve = "3" , twelve ;
twelve thousand two hundred one = twelve , two hundred one ;
```

Expressions that may be omitted or repeated can be represented through curly braces { ... }:

```
natural number = digit excluding zero , { digit } ;
```

In this case, the strings *I*, *2*, ..., *10,...,12345*,... are correct expressions. To represent this, everything that is set within the curly braces may be repeated arbitrarily often, including not at all.

An option can be represented through squared brackets [...]. That is, everything that is set within the square brackets may be present just once, or not at all:

```
integer = "0" | [ "-" ] , natural number ;
```

Therefore an integer is a zero (*0*) or a natural number that may be preceded by an optional minus sign.

EBNF also provides, among other things, the syntax to describe repetitions of a specified number of times, to exclude some part of a production, or to insert comments in an EBNF grammar.

Table of symbols

The following represents a proposed standard.

Usage	Notation
definition	=
concatenation	,
termination	;
alternation	
option	[...]
repetition	{ ... }
grouping	(...)
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?
exception	-

Another example

A Pascal-like programming language that allows only assignments can be defined in EBNF as follows:

```
(* a simple program syntax in EBNF - Wikipedia *)
program = 'PROGRAM' , white space , identifier , white space ,
          'BEGIN' , white space ,
          { assignment , ";" , white space } ,
          'END.' ;
identifier = alphabetic character , { alphabetic character | digit } ;
number = [ "-" ] , digit , { digit } ;
string = '"' , { all characters - '"' } , '"';
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                      | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                      | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                      | "V" | "W" | "X" | "Y" | "Z" ;
```

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

A syntactically correct program then would be:

```
PROGRAM DEMO1
BEGIN
  A0:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A;
  BABOON:=GIRAFFE;
  TEXT:="Hello world!";
END .
```

The language can easily be extended with control flows, arithmetical expressions, and Input/Output instructions. Then a small, usable programming language would be developed.

Advantages of EBNF over BNF

The BNF had the problem that options and repetitions could not be directly expressed. Instead, they needed the use of an intermediate rule or alternative production defined to be either nothing or the optional production for option, or either the repeated production or itself, recursively, for repetition. The same constructs can still be used in EBNF.

The BNF used the symbols (<, >, |, ::=) for itself, but did not include quotes around terminal strings. This prevented these characters from being used in the languages, and required a special symbol for the empty string. In EBNF, terminals are strictly enclosed within quotation marks ("..." or '...'). The angle brackets ("<...>") for nonterminals can be omitted.

BNF syntax can only represent a rule in one line, whereas in EBNF a terminating character, the semicolon, marks the end of a rule.

Furthermore, EBNF includes mechanisms for enhancements, defining the number of repetitions, excluding alternatives, comments, etc.

It should be noted that EBNF is not "more powerful" than the BNF: As a matter of principle, any grammar defined in EBNF can also be represented in BNF (though representations in the latter are generally lengthier).

Conventions

1. The following conventions are used:

- Each meta-identifier of Extended BNF is written as one or more words joined together by hyphens
- A meta-identifier ending with *-symbol* is the name of a terminal symbol of Extended BNF.

2. The normal character representing each operator of Extended BNF and its implied precedence is (highest precedence at the top):

```
* repetition-symbol
- except-symbol
, concatenate-symbol
| definition-separator-symbol
= defining-symbol
```

```
; terminator-symbol
```

3. The normal precedence is overridden by the following bracket pairs:

' first-quote-symbol	first-quote-symbol '
" second-quote-symbol	second-quote-symbol "
(* start-comment-symbol	end-comment-symbol *)
(start-group-symbol	end-group-symbol)
[start-option-symbol	end-option-symbol]
{ start-repeat-symbol	end-repeat-symbol }
? special-sequence-symbol	special-sequence-symbol ?

The first-quote-symbol is the apostrophe as defined by ISO/IEC 646:1991, that is to say Unicode U+0027 ('); the font used in ISO/IEC 14977:1996(E) renders it very much like the acute, Unicode U+00B4 ('), so confusion sometimes arises. However, the ISO Extended BNF standard invokes ISO/IEC 646:1991, "ISO 7-bit coded character set for information interchange", as a normative reference and makes no mention of any other character sets, so formally, there is no confusion with Unicode characters outside the 7-bit ASCII range.

As examples, the following syntax rules illustrate the facilities for expressing repetition:

```
aa = "A";
bb = 3 * aa, "B";
cc = 3 * [aa], "C";
dd = {aa}, "D";
ee = aa, {aa}, "E";
ff = 3 * aa, 3 * [aa], "F";
gg = {3 * aa}, "G";
```

Terminal strings defined by these rules are as follows:

```
aa: A
bb: AAAB
cc: C AC AAC AAAC
dd: D AD AAD AAAD AAAAD etc.
ee: AE AAE AAAE AAAAE AAAAAE etc.
ff: AAAF AAAAF AAAAAF AAAAAAF
gg: G AAAG AAAAAAG etc.
```

EBNF extensibility

According to the ISO 14977 standard EBNF is meant to be extensible, and two facilities are mentioned. The first is part of EBNF grammar, the special sequence, which is arbitrary text enclosed with question marks. The interpretation of the text inside a special sequence is beyond the scope of the EBNF standard. For example, the space character could be defined by the following rule:

```
space = ? US-ASCII character 32 ?;
```

The second facility for extension is using the fact that parentheses cannot in EBNF be placed next to identifiers (they must be concatenated with them). The following is valid EBNF:

```
something = foo, ( bar );
```

The following is *not* valid EBNF:

```
something = foo ( bar );
```

Therefore, an extension of EBNF could use that notation. For example, in a Lisp grammar, function application could be defined by the following rule:

```
function application = list( symbol , { expression } );
```

Related work

- The W3C used a different EBNF^[2] to specify the XML syntax.
- The British Standards Institution published a standard for an EBNF: BS 6154 in 1981.
- The IETF uses Augmented BNF (ABNF), specified in RFC 5234.

References

- Niklaus Wirth: What can we do about the unnecessary diversity of notation for syntactic definitions?^[3] CACM, Vol. 20, Issue 11, November 1977, pp. 822–823.
- Roger S. Scowen: Extended BNF — A generic base standard. Software Engineering Standards Symposium 1993.
- The International standard (ISO 14977^[4]) that defines the EBNF is now freely available as Zip-compressed PDF file^[1].

External links

- Article "EBNF: A Notation to Describe Syntax (PDF)^[5]" by Richard E. Pattis describing the functions and syntax of EBNF
- Article "BNF and EBNF: What are they and how do they work?^[6]" by Lars Marius Garshol
- Article "The Naming of Parts^[7]" by John E. Simpson
- ISO/IEC 14977 : 1996(E)^[8]
- RFC 4234 – Augmented BNF for Syntax Specifications: ABNF
- BNF/EBNF variants^[9] – a table by Pete Jinks comparing several syntaxes.
- Create syntax diagrams from EBNF^[10]
- EBNF Parser & Renderer^[11]

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

References

- [1] [http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip)
- [2] <http://www.w3.org/TR/REC-xml/#sec-notation>
- [3] <http://doi.acm.org/10.1145/359863.359883>
- [4] <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153>
- [5] <http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf>
- [6] <http://www.garshol.priv.no/download/text/bnf.html>
- [7] <http://xml.com/pub/a/2001/07/25/namingparts.html>
- [8] <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>
- [9] <http://www.cs.man.ac.uk/~pjw/bnf/ebnf.html>
- [10] <http://dotnet.jku.at/applications/Visualizer>
- [11] <http://wiki.karmin.ch/ebnf/index>

TBNF

TBNF is an acronym for the phrase, "Translational BNF". BNF refers to Backus Normal Form which is a formal grammar notation used to define the syntax of computer languages, such as Algol, Ada, C++, COBOL, FORTRAN, Java, Perl, Python, and many others. TBNF goes beyond BNF and Extended BNF (EBNF) grammar notation because it not only defines the syntax of a language, but also defines the structure of the abstract-syntax tree (AST) to be created in memory and the output intermediate code to be generated. Thus TBNF defines the complete translation process from input text to intermediate code.

The TBNF concept was first published in April 2006 in a paper at SIGPLAN Notices, a special interest group of the ACM. See the TBNF paper at ACM ^[1].

References

[1] <http://portal.acm.org/citation.cfm?id=1147218>

Top-down parsing

Top-down parsing is a type of parsing strategy wherein one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. LL parsers are a type of parser that uses a top-down parsing strategy.

Top-down parsing is a strategy of analyzing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages.

Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.^[1]

Simple implementations of top-down parsing do not terminate for left-recursive grammars, and top-down parsing with backtracking may have exponential time complexity with respect to the length of the input for ambiguous CFGs.^[2] However, more sophisticated top-down parsers have been created by Frost, Hafiz, and Callaghan^[3] ^[4] which do accommodate ambiguity and left recursion in polynomial time and which generate polynomial-sized representations of the potentially-exponential number of parse trees.

Programming language application

A compiler parses input from a programming language to assembly language or an internal representation by matching the incoming symbols to Backus-Naur form production rules. An LL parser, is a type of parser that does top-down parsing by applying each production rule to the incoming symbols, working from the left-most symbol yielded on a production rule and then proceeding to the next production rule for each non-terminal symbol encountered. In this way the parsing starts on the Left of the result side (right side) of the production rule and evaluates non-terminals from the Left first and, thus, proceeds down the parse tree for each new non-terminal before continuing to the next symbol for a production rule.

For example:

- $A \rightarrow aBC$
- $B \rightarrow c|cd$
- $C \rightarrow df|eg$

would match $A \rightarrow aBC$ and attempt to match $B \rightarrow c|cd$ next. Then $C \rightarrow df|eg$ would be tried. As one may expect, some languages are more ambiguous than others. For a non-ambiguous language in which all productions for a non-terminal produce distinct strings: the string produced by one production will not start with the same symbol as the string produced by another production. A non-ambiguous language may be parsed by an LL(1) grammar where the (1) signifies the parser reads ahead one token at a time. For an ambiguous language to be parsed by an LL parser, the parser must lookahead more than 1 symbol, e.g. LL(3).

The common solution to this problem is to use an LR parser, which is a type of shift-reduce parser, and does bottom-up parsing.

Accommodating left recursion in top-down parsing

A formal grammar that contains left recursion cannot be parsed by a naive recursive descent parser unless they are converted to a weakly equivalent right-recursive form. However, recent research demonstrates that it is possible to accommodate left-recursive grammars (along with all other forms of general CFGs) in a more sophisticated top-down parser by use of curtailment. A recognition algorithm which accommodates ambiguous grammars and curtails an ever-growing direct left-recursive parse by imposing depth restrictions with respect to input length and current input position, is described by Frost and Hafiz in 2006.^[5] That algorithm was extended to a complete parsing algorithm to accommodate indirect (by comparing previously-computed context with current context) as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially-exponential number of parse trees for highly-ambiguous grammars by Frost, Hafiz and Callaghan in 2007.^[3] The algorithm has since been implemented as a set of parser combinators written in the Haskell programming language. The implementation details of these new set of combinators can be found in a paper^[4] by the above-mentioned authors, which was presented in PADL'08. The X-SAIGA^[7] site has more about the algorithms and implementation details.

Time and space complexity of top-down parsing

When top-down parser tries to parse an ambiguous input with respect to an ambiguous CFG, it may need exponential number of steps (with respect to the length of the input) to try all alternatives of the CFG in order to produce all possible parse trees, which eventually would require exponential memory space. The problem of exponential time complexity in top-down parsers constructed as sets of mutually-recursive functions has been solved by Norvig in 1991.^[6] His technique is similar to the use of dynamic programming and state-sets in Earley's algorithm (1970), and tables in the CYK algorithm of Cocke, Younger and Kasami.

The key idea is to store results of applying a parser p_j at position j in a memotable and to reuse results whenever the same situation arises. Frost, Hafiz and Callaghan^{[3] [4]} also use memoization for refraining redundant computations to accommodate any form of CFG in polynomial time ($\Theta(n^4)$ for left-recursive grammars and $\Theta(n^3)$ for non left-recursive grammars). Their top-down parsing algorithm also requires polynomial space for potentially exponential ambiguous parse trees by 'compact representation' and 'local ambiguities grouping'. Their compact representation is comparable with Tomita's compact representation of bottom-up parsing.^[7]

Using PEG's, another representation of grammars, packrat parsers provide an elegant and powerful parsing algorithm. See Parsing expression grammar.

References

- [1] Aho, A.V., Sethi, R. and Ullman ,J.D. (1986) " Compilers: principles, techniques, and tools." *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.*
- [2] Aho, A.V and Ullman, J. D. (1972) " The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing." *Englewood Cliffs, N.J.: Prentice-Hall.*
- [3] Frost, R., Hafiz, R. and Callaghan, P. (2007) " Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109 - 120, June 2007, Prague.
- [4] Frost, R., Hafiz, R. and Callaghan, P. (2008) " Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902/2008, Pages: 167-181, January 2008, San Francisco.
- [5] Frost, R. and Hafiz, R. (2006) " A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time." *ACM SIGPLAN Notices*, Volume 41 Issue 5, Pages: 46 - 54.
- [6] Norvig, P. (1991) "Techniques for automatic memoisation with applications to context-free parsing." *Journal - Computational Linguistics* Volume 17, Issue 1, Pages: 91 - 98.
- [7] Tomita, M. (1985) "Efficient Parsing for Natural Language." *Kluwer, Boston, MA.*

External links

- X-SAIGA (<http://www.cs.uwindsor.ca/~hafiz/proHome.html>) - eXecutable SpecificAtIons of GrAmmars

Recursive descent parser

A **recursive descent parser** is a top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

A **predictive parser** is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. (The LL(k) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(k) grammar.) A predictive parser runs in linear time.

Recursive descent with backup is a technique that determines which production to use by trying each production in turn. Recursive descent with backup is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k). Even when they terminate, parsers that use recursive descent with backup may require exponential time.

Although predictive parsers are widely used, programmers often prefer to create LR or LALR parsers via parser generators without transforming the grammar into LL(k) form.

Example parser

The following EBNF-like grammar (for Niklaus Wirth's PL/0 programming language, from *Algorithms + Data Structures = Programs*) is in LL(1) form:

```
program = block "."
block =
  ["const" ident "=" number {"," ident "=" number} ";" ]
  ["var" ident {"," ident} ";" ]
  {"procedure" ident ";" block ";" } statement .
```

```

statement =
  [ident ":=" expression
  | "call" ident
  | "begin" statement {";" statement} "end"
  | "if" condition "then" statement
  | "while" condition "do" statement
  ] .

condition =
  "odd" expression
  | expression ("=" | "#" | "<" | "<=" | ">" | ">=") expression .

expression = ["+" | "-"] term {("+" | "-") term} .

term = factor {("*" | "/") factor} .

factor =
  ident
  | number
  | "(" expression ")" .

```

Terminals are expressed in quotes. Each nonterminal is defined by a rule in the grammar, except for *ident* and *number*, which are assumed to be implicitly defined.

C implementation

What follows is an implementation of a recursive descent parser for the above language in C. The parser reads in source code, and exits with an error message if the code fails to parse, exiting silently if the code parses correctly.

Notice how closely the predictive parser below mirrors the grammar above. There is a procedure for each nonterminal in the grammar. Parsing descends in a top-down manner, until the final nonterminal has been processed. The program fragment depends on a global variable, *sym*, which contains the next symbol from the input, and the function *getsym*, which updates *sym* when called.

The implementations of the functions *getsym* and *error* are omitted for simplicity.

```

typedef enum {ident, number, lparen, rparen, times, slash, plus,
  minus, eql, neq, lss, leq, gtr, geq, callsym, beginsym, semicolon,
  endsym, ifsym, whilesym, becomes, thensym, dosym, constsym, comma,
  varsym, procsym, period, oddsym} Symbol;

Symbol sym;
void getsym(void);
void error(const char msg[]);
void expression(void);

int accept(Symbol s) {
  if (sym == s) {
    getsym();
    return 1;
}

```

```
        }
        return 0;
    }

int expect(Symbol s) {
    if (accept(s))
        return 1;
    error("expect: unexpected symbol");
    return 0;
}

void factor(void) {
    if (accept(ident)) {
        ;
    } else if (accept(number)) {
        ;
    } else if (accept(lparen)) {
        expression();
        expect(rparen);
    } else {
        error("factor: syntax error");
        getsym();
    }
}

void term(void) {
    factor();
    while (sym == times || sym == slash) {
        getsym();
        factor();
    }
}

void expression(void) {
    if (sym == plus || sym == minus)
        getsym();
    term();
    while (sym == plus || sym == minus) {
        getsym();
        term();
    }
}

void condition(void) {
    if (accept(oddssym)) {
        expression();
    } else {
```

```
expression();
if (sym == eql || sym == neq || sym == lss || sym == leq || sym
== gtr || sym == geq) {
    getsym();
    expression();
} else {
    error("condition: invalid operator");
    getsym();
}
}

void statement(void) {
if (accept(ident)) {
    expect(becomes);
    expression();
} else if (accept(callsym)) {
    expect(ident);
} else if (accept(beginsym)) {
    do {
        statement();
    } while (accept(semicolon));
    expect(endsym());
} else if (accept(ifsym)) {
    condition();
    expect(thensym);
    statement();
} else if (accept(whilesym)) {
    condition();
    expect(dosym);
    statement();
}
}

void block(void) {
if (accept(constsym)) {
    do {
        expect(ident);
        expect(eql);
        expect(number);
    } while (accept(comma));
    expect(semicolon);
}
if (accept(varsym)) {
    do {
        expect(ident);
    } while (accept(comma));
}
```

```
        expect (semicolon) ;
    }
    while (accept (procsym) ) {
        expect (ident) ;
        expect (semicolon) ;
        block () ;
        expect (semicolon) ;
    }
    statement () ;
}

void program(void) {
    getsym () ;
    block () ;
    expect (period) ;
}
```

Implementation in functional languages

Recursive descent parsers are particularly easy to implement in functional languages such as Haskell, Lisp or ML as they tend to be better suited for recursion in general.

See Functional Pearls: Monadic Parsing in Haskell ^[1]

References

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

- *Compilers: Principles, Techniques, and Tools*, first edition, Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman, in particular Section 4.4.
- *Modern Compiler Implementation in Java, Second Edition*, Andrew Appel, 2002, ISBN 0-521-82060-X.
- *Recursive Programming Techniques*, W.H. Burge, 1975, ISBN 0-201-14450-6
- *Crafting a Compiler with C*, Charles N Fischer and Richard J LeBlanc, Jr, 1991, ISBN 0-8053-2166-7.
- Parse::RecDescent ^[2]: A versatile recursive descent Perl module.
- pyparsing ^[3]: A versatile Python recursive parsing module that is not recursive descent (python-list post ^[4]).
- Jparsec ^[5] a Java port of Haskell's Parsec module.
- *Compiling with C# and Java*, Pat Terry, 2005, ISBN 0-321-26360-X, 624
- *Algorithms + Data Structures = Programs*, Niklaus Wirth, 1975, ISBN 0-13-022418-9
- *Compiler Construction*, Jonatan Rugarn, 1996, ISBN 0-201-40353-6

External links

- Introduction to Parsing [6] - an easy to read introduction to parsing, with a comprehensive section on recursive descent parsing
- How to turn a Grammar into C code [7] - a brief tutorial on implementing recursive descent parser
- Simple mathematical expressions parser [8] in Ruby
- Jack W. Crenshaw: *Let's Build A Compiler* (1988-1995) [3], in Pascal, with assembler output, using a "keep it simple" approach

References

- [1] <http://www.cs.nott.ac.uk/~gmh/pearl.pdf>
- [2] <http://search.cpan.org/~dconway/Parse-RecDescent-1.94/lib/Parse/RecDescent.pod>
- [3] <http://pyparsing.sourceforge.net/>
- [4] <http://mail.python.org/pipermail/python-list/2007-November/636291.html>
- [5] <http://jparsec.codehaus.org/>
- [6] <http://www.mollypages.org/page/grammar/index.mp>
- [7] http://teaching.idallen.com/cst8152/98w/recursive_descent_parsing.html
- [8] <http://lukaszwrobel.pl/blog/math-parser-part-3-implementation>

Tail recursive parser

Tail recursive parsers are derived from the more common Recursive descent parsers. Tail recursive parsers are commonly used to parse left recursive grammars. They use a smaller amount of stack space than regular recursive descent parsers. They are also easy to write. Typical recursive descent parsers make parsing left recursive grammars impossible (because of an infinite loop problem). Tail recursive parsers use a node reparenting technique that makes this allowable. Given an EBNF Grammar such as the following:

```
E : T
T: T ('+' F)* | F
F: F ('*' I)* | I
I: <identifier>
```

A simple tail recursive parser can be written much like a recursive descent parser. The typical algorithm for parsing a grammar like this using an Abstract syntax tree is:

- Parse the next level of the grammar and get its output tree, designate it the first tree, F
- While there is terminating token, T, that can be put as the parent of this node:
 - Allocate a new node, N
 - Set N's current operator as the current input token
 - Advance the input one token
 - Set N's left subtree as F
 - Parse another level down again and store this as the next tree, X
 - Set N's right subtree as X
 - Set F to N
- Return N

A C programming language implementation of this parser is shown here:

```
typedef struct _exptree exptree;
struct _exptree {
    char token;
```

```
    exptree *left;
    exptree *right;
}

exptree *parse_e(void)
{
    return parse_t();
}

exptree *parse_t(void)
{
    exptree *first_f = parse_f();

    while(cur_token() == '+') {
        exptree *replace_tree = alloc_tree();
        replace_tree->token = cur_token();
        replace_tree->left = first_f;
        next_token();
        replace_tree->right = parse_f();
        first_f = replace_tree;
    }

    return first_f;
}

exptree *parse_f(void)
{
    exptree *first_i = parse_i();

    while(cur_token() == '*') {
        exptree *replace_tree = alloc_tree();
        replace_tree->token = cur_token();
        replace_tree->left = first_i;
        next_token();
        replace_tree->right = parse_i();
        first_i = replace_tree;
    }

    return first_i;
}

exptree *parse_i(void)
{
    exptree *i = alloc_tree();
    exptree->left = exptree->right = NULL;
    exptree->token = cur_token();
    next_token();
}
```

```

    return i;
}

```

Further reading

- Article in the January 2006 edition of Dr. Dobbs Journal, "Recursive Descent, Tail Recursion, & the Dreaded Double Divide" [1]

References

- [1] <http://www.drdobbs.com/cpp/184406384?pgno=1>

Parsing expression grammar

A **parsing expression grammar**, or **PEG**, is a type of analytic formal grammar, i.e. it describes a formal language in terms of a set of rules for recognizing strings in the language. The formalism was introduced by Bryan Ford in 2004^[1] and is closely related to the family of top-down parsing languages introduced in the early 1970s. Syntactically, PEGs also look similar to context-free grammars (CFGs), but they have a different interpretation: the rules of PEGs are ordered. This is closer to how string recognition tends to be done in practice, e.g. by a recursive descent parser.

Unlike CFGs, PEGs cannot be ambiguous; if a string parses, it has exactly one valid parse tree. This makes PEGs well-suited to parsing computer languages, but not natural languages.

Definition

Syntax

Formally, a parsing expression grammar consists of:

- A finite set N of *nonterminal symbols*.
- A finite set Σ of *terminal symbols* that is disjoint from N .
- A finite set P of *parsing rules*.
- An expression e_S termed the *starting expression*.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a *parsing expression*. A parsing expression is a hierarchical expression similar to a regular expression, which is constructed in the following fashion:

1. An *atomic parsing expression* consists of:

- any terminal symbol,
- any nonterminal symbol, or
- the empty string ϵ .

2. Given any existing parsing expressions e , e_1 , and e_2 , a new parsing expression can be constructed using the following operators:

- *Sequence*: $e_1 e_2$
- *Ordered choice*: e_1 / e_2
- *Zero-or-more*: e^*
- *One-or-more*: e^+
- *Optional*: $e?$
- *And-predicate*: $&e$
- *Not-predicate*: $!e$

Semantics

The fundamental difference between context-free grammars and parsing expression grammars is that the PEG's choice operator is *ordered*. If the first alternative succeeds, the second alternative is ignored. Thus ordered choice is not commutative, unlike unordered choice as in context-free grammars and regular expressions. Ordered choice is analogous to soft cut operators available in some logic programming languages.

The consequence is that if a CFG is transliterated directly to a PEG, any ambiguity in the former is resolved by deterministically picking one parse tree from the possible parses. By carefully choosing the order in which the grammar alternatives are specified, a programmer has a great deal of control over which parse tree is selected.

Like boolean context-free grammars, parsing expression grammars also add the and- and not- predicates. These facilitate further disambiguation, if reordering the alternatives cannot specify the exact parse tree desired.

Operational interpretation of parsing expressions

Each nonterminal in a parsing expression grammar essentially represents a parsing function in a recursive descent parser, and the corresponding parsing expression represents the "code" comprising the function. Each parsing function conceptually takes an input string as its argument, and yields one of the following results:

- *success*, in which the function may optionally move forward or *consume* one or more characters of the input string supplied to it, or
- *failure*, in which case no input is consumed.

A nonterminal may succeed without actually consuming any input, and this is considered an outcome distinct from failure.

An atomic parsing expression consisting of a single terminal succeeds if the first character of the input string matches that terminal, and in that case consumes the input character; otherwise the expression yields a failure result.

An atomic parsing expression consisting of the empty string always trivially succeeds without consuming any input.

An atomic parsing expression consisting of a nonterminal A represents a recursive call to the nonterminal-function A .

The sequence operator $e_1 \ e_2$ first invokes e_1 , and if e_1 succeeds, subsequently invokes e_2 on the remainder of the input string left unconsumed by e_1 , and returns the result. If either e_1 or e_2 fails, then the sequence expression $e_1 \ e_2$ fails.

The choice operator e_1 / e_2 first invokes e_1 , and if e_1 succeeds, returns its result immediately. Otherwise, if e_1 fails, then the choice operator backtracks to the original input position at which it invoked e_1 , but then calls e_2 instead, returning e_2 's result.

The zero-or-more, one-or-more, and optional operators consume zero or more, one or more, or zero or one consecutive repetitions of their sub-expression e , respectively. Unlike in context-free grammars and regular expressions, however, these operators *always* behave greedily, consuming as much input as possible and never backtracking. (Regular expression matchers may start by matching greedily, but will then backtrack and try shorter matches if they fail to match.) For example, the expression a^* will always consume as many a's as are consecutively available in the input string, and the expression $(a^* a)$ will always fail because the first part (a^*) will never leave any a's for the second part to match.

Finally, the and-predicate and not-predicate operators implement syntactic predicates. The expression $\&e$ invokes the sub-expression e , and then succeeds if e succeeds and fails if e fails, but in either case *never consumes any input*. Conversely, the expression $!e$ succeeds if e fails and fails if e succeeds, again consuming no input in either case. Because these can use an arbitrarily complex sub-expression e to "look ahead" into the input string without actually consuming it, they provide a powerful syntactic lookahead and disambiguation facility.

Examples

This is a PEG that recognizes mathematical formulas that apply the basic four operations to non-negative integers.

```
Value   ← [0-9]+ / (' Expr ')
Product ← Value (('*' / '/') Value)*
Sum     ← Product (('+' / '-') Product)*
Expr    ← Sum
```

In the above example, the terminal symbols are characters of text, represented by characters in single quotes, such as `'('` and `')'`. The range `[0-9]` is also a shortcut for ten characters, indicating any one of the digits 0 through 9. (This range syntax is the same as the syntax used by regular expressions.) The nonterminal symbols are the ones that expand to other rules: `Value`, `Product`, `Sum`, and `Expr`.

The examples below drop quotation marks in order to be easier to read. Lowercase letters are terminal symbols, while capital letters in italics are nonterminals. Actual PEG parsers would require the lowercase letters to be in quotes.

The parsing expression `(a/b)*` matches and consumes an arbitrary-length sequence of a's and b's. The rule `S ← a S?` **b** describes the simple context-free "matching language" $\{a^n b^n : n \geq 1\}$. The following parsing expression grammar describes the classic non-context-free language $\{a^n b^n c^n : n \geq 1\}$:

```
S ← &(A c) a+ B !(a/b/c)
A ← a A? b
B ← b B? c
```

The following recursive rule matches standard C-style if/then/else statements in such a way that the optional "else" clause always binds to the innermost "if", because of the implicit prioritization of the `/` operator. (In a context-free grammar, this construct yields the classic dangling else ambiguity.)

```
S ← if C then S else S / if C then S
```

The parsing expression `foo &(bar)` matches and consumes the text "foo" but only if it is followed by the text "bar". The parsing expression `foo !(bar)` matches the text "foo" but only if it is *not* followed by the text "bar". The expression `!(a+b) a` matches a single "a" but only if it is not the first in an arbitrarily-long sequence of a's followed by a b.

The following recursive rule matches Pascal-style nested comment syntax, (* which can (* nest *) like this *). The comment symbols appear in double quotes to distinguish them from PEG operators.

```
Begin ← "(*"
End   ← "*)"
C     ← Begin N* End
N     ← C / (!Begin !End Z)
Z     ← any single character
```

Implementing parsers from parsing expression grammars

Any parsing expression grammar can be converted directly into a recursive descent parser.^[2] Due to the unlimited lookahead capability that the grammar formalism provides, however, the resulting parser could exhibit exponential time performance in the worst case.

It is possible to obtain better performance for any parsing expression grammar by converting its recursive descent parser into a *packrat parser*, which always runs in linear time, at the cost of substantially greater storage space requirements. A packrat parser^[2] is a form of parser similar to a recursive descent parser in construction, except that during the parsing process it memoizes the intermediate results of all invocations of the mutually recursive parsing functions, ensuring that each parsing function is only invoked at most once at a given input position. Because of this memoization, a packrat parser has the ability to parse many context-free grammars and *any* parsing expression grammar (including some that do not represent context-free languages) in linear time. Examples of memoized recursive descent parsers are known from at least as early as 1993.^[3] Note that this analysis of the performance of a packrat parser assumes that enough memory is available to hold all of the memoized results; in practice, if there were not enough memory, some parsing functions might have to be invoked more than once at the same input position, and consequently the parser could take more than linear time.

It is also possible to build LL parsers and LR parsers from parsing expression grammars, with better worst-case performance than a recursive descent parser, but the unlimited lookahead capability of the grammar formalism is then lost. Therefore, not all languages that can be expressed using parsing expression grammars can be parsed by LL or LR parsers.

Advantages

Compared to pure regular expressions (i.e. without back-references), PEGs are strictly more powerful, but require significantly more memory. For example, a regular expression inherently cannot find an arbitrary number of matched pairs of parentheses, because it is not recursive, but a PEG can. However, a PEG will require an amount of memory proportional to the length of the input, while a regular expression matcher will require only a constant amount of memory.

Any PEG can be parsed in linear time by using a packrat parser, as described above.

Parsers for languages expressed as a CFG, such as LR parsers, require a separate tokenization step to be done first, which breaks up the input based on the location of spaces, punctuation, etc. The tokenization is necessary because of the way these parsers use *lookahead* to parse CFGs that meet certain requirements in linear time. PEGs do not require tokenization to be a separate step, and tokenization rules can be written in the same way as any other grammar rule.

Many CFGs contain inherent ambiguities, even when they're intended to describe unambiguous languages. The "dangling else" problem in C, C++, and Java is one example. These problems are often resolved by applying a rule outside of the grammar. In a PEG, these ambiguities never arise, because of prioritization.

Disadvantages

Memory consumption

PEG parsing is typically carried out via *packrat parsing*, which uses memoization^{[4] [5]} to eliminate redundant parsing steps. Packrat parsing requires storage proportional to the total input size, rather than the depth of the parse tree as with LR parsers. This is a significant difference in many domains: for example, hand-written source code has an effectively constant expression nesting depth independent of the length of the program—expressions nested beyond a certain depth tend to get refactored.

For some grammars and some inputs, the depth of the parse tree can be proportional to the input size,^[6] so that in asymptotic the worst-case which does not distinguish these two metrics packrat parsing will appear to be no worse than an LR parser. This is similar to a situation which arises in graph algorithms: the Bellman–Ford algorithm and Floyd–Warshall algorithm appear to have the same running time ($O(|V|^3)$) if only the number of vertices is considered. However, a more accurate analysis which accounts for the number of edges as a separate parameter assigns the Bellman–Ford algorithm a time of $O(|V| * |E|)$, which is only quadratic in the size of the input (rather than cubic).

Indirect left recursion

PEGs cannot express *left-recursive* rules where a rule refers to itself without moving forward in the string. For example, in the arithmetic grammar above, it would be tempting to move some rules around so that the precedence order of products and sums could be expressed in one line:

```
Value   ← [0-9.] + / '(' Expr ')'
Product ← Expr (('*' / '/') Expr) *
Sum     ← Expr (( '+' / '-' ) Expr) *
Expr    ← Product / Sum / Value
```

In this new grammar matching an `Expr` requires testing if a `Product` matches while matching a `Product` requires testing if an `Expr` matches. This circular definition cannot be resolved. However, left-recursive rules can always be rewritten to eliminate left-recursion. For example, the following left-recursive CFG rule:

```
string-of-a ← string-of-a 'a' | 'a'
```

can be rewritten in a PEG using the plus operator:

```
string-of-a ← 'a'+
```

The process of rewriting *indirectly* left-recursive rules is complex in some packrat parsers, especially when semantic actions are involved.

With some modification, traditional packrat parsing can support direct left recursion.^{[2] [7] [8]} but doing so results in a loss of the linear-time parsing property^[7] which is generally the justification for using PEGs and Packrat Parsing in the first place. Only the OMeta parsing algorithm^[9] supports full direct and indirect left recursion without additional attendant complexity (but again, at a loss of the linear time complexity), whereas all GLR parsers support left recursion.

Subtle grammatical errors

In order to express a grammar as a PEG, the grammar author must convert all instances of nondeterministic choice into prioritized choice. Unfortunately, this process is error prone, and often results in grammars which mis-parse certain inputs. An example and commentary can be found here^[10].

Expressive power

Packrat parsers cannot recognize some unambiguous grammars, such as the following (example taken from^[11])

```
S ← 'x' S 'x' | 'x'
```

In fact, neither LL(k) nor LR(k) parsing algorithms are capable of recognizing this example.

Maturity

PEGs are new and not widely implemented. By contrast, regular expressions and CFGs have been around for decades, the code to parse them has been extensively optimized, and many programmers are familiar with how to use them. This is not all that compelling an objection to the adoption and use of PEGs, of course, since there was a time when the same considerations applied to the older approaches as well.

References

- [1] [Ford, Bryan p (<http://www.brynosaurus.com/>)] (2004). "Parsing Expression Grammars: A Recognition Based Syntactic Foundation" (<http://portal.acm.org/citation.cfm?id=964001.964011>). *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. doi:10.1145/964001.964011. ISBN 1-58113-729-X. . Retrieved 2010-07-30.
- [2] Ford, Bryan (September 2002). "Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking" (<http://pdos.csail.mit.edu/~baford/packrat/thesis>). Massachusetts Institute of Technology. . Retrieved 2007-07-27.
- [3] Merritt, Doug (November 1993). "Transparent Recursive Descent" (<http://compilers.iecc.com/comparch/article/93-11-012>). Usenet group comp.compilers. . Retrieved 2009-09-04.
- [4] Bryan Ford. "The Packrat Parsing and Parsing Expression Grammars Page" (<http://pdos.csail.mit.edu/~baford/packrat/>). . Retrieved 23 Nov 2010.
- [5] Rick Jelliffe. "What is a Packrat Parser? What are Brzozowski Derivatives?" (<http://broadcast.oreilly.com/2010/03/what-is-a-packrat-parser-what.html>). . Retrieved 23 Nov 2010.
- [6] for example, the LISP expression (x (x (x (x))))
- [7] Alessandro Warth, James R. Douglass, Todd Millstein (January 2008) (PDF). *Packrat Parsers Can Support Left Recursion* (http://www.vpri.org/pdf/tr2007002_packrat.pdf). Viewpoints Research Institute. . Retrieved 2008-10-02.
- [8] Ruedi Steinmann (March 2009) (PDF). *Handling Left Recursion in Packrat Parsers* (<http://n.ethz.ch/~ruediste/packrat.pdf>). .
- [9] "Packrat Parsers Can Support Left Recursion" (http://www.vpri.org/pdf/tr2007002_packrat.pdf). PEPM '08. January 2008. . Retrieved 2009-08-04.
- [10] <http://article.gmane.org/gmane.comp.parsers.peg.general/2>
- [11] Bryan Ford (2002) (PDF). *Functional Pearl: Packrat Parsing: Simple, Powerful, Lazy, Linear Time* (<http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat/icfp02.pdf>). .
- Medeiros, Sérgio; Ierusalimschy, Roberto (2008). "A parsing machine for PEGs". *Proc. of the 2008 symposium on Dynamic languages*. ACM. pp. article #2. doi:10.1145/1408681.1408683. ISBN 978-1-60558-270-2.

External links

- Parsing Expression Grammars: A Recognition-Based Syntactic Foundation (<http://www.brynosaurus.com/pub/lang/peg-slides.pdf>) (PDF slides)
- The Packrat Parsing and Parsing Expression Grammars Page (<http://pdos.csail.mit.edu/~baford/packrat/>)
- Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking (<http://pdos.csail.mit.edu/~baford/packrat/thesis/>)
- The constructed language Lojban has a fairly large PEG grammar (<http://www.digitalkingdom.org/~rlpowell/hobbies/lojban/grammar/>) allowing unambiguous parsing of Lojban text.
- The Aurochs parser generator has an on-line parsing demo (<http://aurochs.fr/cgi/demo.cgi>) that displays the parse tree for any given grammar and input
- REBOL parse dialect is PEG-compatible
- Parboiled (<https://github.com/sirthias/parboiled/wiki>) is a mixed Java/Scala library for building PEG-based parsers
- Mouse (<http://mousepeg.sourceforge.net>) transcribes PEG into a set of recursive Java procedures

LL parser

An **LL parser** is a top-down parser for a subset of the context-free grammars. It parses the input from **Left** to right, and constructs a **Leftmost** derivation of the sentence (hence LL, compared with LR parser). The class of grammars which are parsable in this way is known as the *LL grammars*.

The remainder of this article describes the table-based kind of parser, the alternative being a recursive descent parser which is usually coded by hand (although not always; see e.g. ANTLR for an LL(*) recursive-descent parser generator).

An LL parser is called an LL(k) parser if it uses k tokens of lookahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(k) grammar. A language that has an LL(k) grammar is known as an LL(k) language. There are LL($k+n$) languages that are not LL(k) languages.^[1] A corollary of this is that not all context-free languages are LL(k) languages.

LL(1) grammars are very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions. Languages based on grammars with a high value of k have traditionally been considered to be difficult to parse, although this is less true now given the availability and widespread use of parser generators supporting LL(k) grammars for arbitrary k .

An LL parser is called an LL(*) parser if it is not restricted to a finite k tokens of lookahead, but can make parsing decisions by recognizing whether the following tokens belong to a regular language (for example by use of a Deterministic Finite Automaton).

General case

The parser works on strings from a particular context-free grammar.

The parser consists of

- an *input buffer*, holding the input string (built from the grammar)
- a *stack* on which to store the terminals and non-terminals from the grammar yet to be parsed
- a *parsing table* which tells it what (if any) grammar rule to apply given the symbols on top of its stack and the next input token

The parser applies the rule found in the table by matching the top-most symbol on the stack (row) with the current symbol in the input stream (column).

When the parser starts, the stack already contains two symbols:

```
[ S, $ ]
```

where '\$' is a special terminal to indicate the bottom of the stack and the end of the input stream, and 'S' is the start symbol of the grammar. The parser will attempt to rewrite the contents of this stack to what it sees on the input stream. However, it only keeps on the stack what still needs to be rewritten.

Concrete example

Set up

To explain its workings we will consider the following small grammar:

1. $S \rightarrow F$
2. $S \rightarrow (S + F)$
3. $F \rightarrow a$

and parse the following input:

(a + a)

The parsing table for this grammar looks as follows:

		()	a	+	\$
S	2	-	1	-	-	
F	-	-	3	-	-	

(Note that there is also a column for the special terminal, represented here as \$, that is used to indicate the end of the input stream.)

Parsing procedure

In each step, the parser reads the next-available symbol from the input stream, and the top-most symbol from the stack. If the input symbol and the stack-top symbol match, the parser discards them both, leaving only the unmatched symbols in the input stream and on the stack.

Thus, in its first step, the parser reads the input symbol '(' and the stack-top symbol 'S'. The parsing table instruction comes from the column headed by the input symbol '(' and the row headed by the stack-top symbol 'S'; this cell contains '2', which instructs the parser to apply rule (2). The parser has to rewrite 'S' to '(S + F)' on the stack and write the rule number 2 to the output. The stack then becomes:

[(, S, +, F,), \$]

Since the '(' from the input stream did not match the top-most symbol, 'S', from the stack, it was not removed, and remains the next-available input symbol for the following step.

In the second step, the parser removes the '(' from its input stream and from its stack, since they match. The stack now becomes:

[S, +, F,), \$]

Now the parser has an 'a' on its input stream and an 'S' as its stack top. The parsing table instructs it to apply rule (1) from the grammar and write the rule number 1 to the output stream. The stack becomes:

[F, +, F,), \$]

The parser now has an 'a' on its input stream and an 'F' as its stack top. The parsing table instructs it to apply rule (3) from the grammar and write the rule number 3 to the output stream. The stack becomes:

[a, +, F,), \$]

In the next two steps the parser reads the 'a' and '+' from the input stream and, since they match the next two items on the stack, also removes them from the stack. This results in:

[F,), \$]

In the next three steps the parser will replace 'F' on the stack by 'a', write the rule number 3 to the output stream and remove the 'a' and ')' from both the stack and the input stream. The parser thus ends with '\$' on both its stack and its input stream.

In this case the parser will report that it has accepted the input string and write the following list of rule numbers to the output stream:

[2, 1, 3, 3]

This is indeed a list of rules for a leftmost derivation of the input string, which is:

$S \rightarrow (S + F) \rightarrow (F + F) \rightarrow (a + F) \rightarrow (a + a)$

Parser implementation in C++

Below follows a C++ implementation of a table-based LL parser for the example language:

```
#include <iostream>
#include <map>
#include <stack>

enum Symbols {
    // the symbols:
    // Terminal symbols:
    TS_L_PARENS,      // (
    TS_R_PARENS,      // )
    TS_A,              // a
    TS_PLUS,           // +
    TS_EOS,            // $, in this case corresponds to '\0'
    TS_INVALID,        // invalid token

    // Non-terminal symbols:
    NTS_S,             // S
    NTS_F
};

/*
Converts a valid token to the corresponding terminal symbol
*/
enum Symbols lexer(char c)
{
    switch(c)
    {
        case '(':
            return TS_L_PARENS;
            break;

        case ')':
            return TS_R_PARENS;
            break;

        case 'a':
            return TS_A;
            break;
    }
}
```

```
        return TS_A;
        break;

    case '+':
        return TS_PLUS;
        break;

    case '\0':      // this will act as the $ terminal symbol
        return TS_EOS;
        break;

    default:
        return TS_INVALID;
        break;
    }
}

int main(int argc, char **argv)
{
    using namespace std;

    if (argc < 2)
    {
        cout << "usage:\n\tll '(a+a)'" << endl;
        return 0;
    }

    map< enum Symbols, map<enum Symbols, int> > table; // LL parser table, maps < non-terminal, terminal> pair to action
    stack<enum Symbols> ss;      // symbol stack
    char *p;      // input buffer

    // initialize the symbols stack
    ss.push(TS_EOS);      // terminal, $
    ss.push(NTS_S);       // non-terminal, S

    // initialize the symbol stream cursor
    p = &argv[1][0];

    // setup the parsing table
    table[NTS_S][TS_L_PARENS] = 2;      table[NTS_S][TS_A] = 1;
    table[NTS_F][TS_A] = 3;

    while(ss.size() > 0)
    {
        if(lexer(*p) == ss.top())
        {
            cout << "Matched symbols: " << lexer(*p) << endl;
        }
    }
}
```

```
p++;

ss.pop();

}

else

{

    cout << "Rule " << table[ss.top()][lexer(*p)] << endl;

    switch(table[ss.top()][lexer(*p)])

    {

        case 1:      // 1. S → F

            ss.pop();

            ss.push(NTS_F);      // F

            break;


        case 2:      // 2. S → ( S + F )

            ss.pop();

            ss.push(TS_R_PARENS);      // )

            ss.push(NTS_F);      // F

            ss.push(TS_PLUS);      // +

            ss.push(NTS_S);      // S

            ss.push(TS_L_PARENS);      // (

            break;


        case 3:      // 3. F → a

            ss.pop();

            ss.push(TS_A);      // a

            break;


        default:

            cout << "parsing table defaulted" << endl;

            return 0;

            break;

    }

}

cout << "finished parsing" << endl;

return 0;
}
```

Remarks

As can be seen from the example the parser performs three types of steps depending on whether the top of the stack is a nonterminal, a terminal or the special symbol \$:

- If the top is a nonterminal then it looks up in the parsing table on the basis of this nonterminal and the symbol on the input stream which rule of the grammar it should use to replace it with on the stack. The number of the rule is written to the output stream. If the parsing table indicates that there is no such rule then it reports an error and stops.
- If the top is a terminal then it compares it to the symbol on the input stream and if they are equal they are both removed. If they are not equal the parser reports an error and stops.
- If the top is \$ and on the input stream there is also a \$ then the parser reports that it has successfully parsed the input, otherwise it reports an error. In both cases the parser will stop.

These steps are repeated until the parser stops, and then it will have either completely parsed the input and written a leftmost derivation to the output stream or it will have reported an error.

Constructing an LL(1) parsing table

In order to fill the parsing table, we have to establish what grammar rule the parser should choose if it sees a nonterminal A on the top of its stack and a symbol a on its input stream. It is easy to see that such a rule should be of the form $A \rightarrow w$ and that the language corresponding to w should have at least one string starting with a . For this purpose we define the *First-set* of w , written here as $\text{Fi}(w)$, as the set of terminals that can be found at the start of some string in w , plus ϵ if the empty string also belongs to w . Given a grammar with the rules $A_1 \rightarrow w_1, \dots, A_n \rightarrow w_n$, we can compute the $\text{Fi}(w_i)$ and $\text{Fi}(A_i)$ for every rule as follows:

1. initialize every $\text{Fi}(w_i)$ and $\text{Fi}(A_i)$ with the empty set
2. add $F_i(w_i)$ to $\text{Fi}(A_i)$ for every rule $A_i \rightarrow w_i$, where F_i is defined as follows:
 - $F_i(a w') = \{ a \}$ for every terminal a
 - $F_i(A w') = \text{Fi}(A) \cup F_i(w')$ for every nonterminal A with ϵ not in $\text{Fi}(A)$
 - $F_i(A w') = \text{Fi}(A) \setminus \{ \epsilon \} \cup F_i(w')$ for every nonterminal A with ϵ in $\text{Fi}(A)$
 - $F_i(\epsilon) = \{ \epsilon \}$
3. add $\text{Fi}(w_i)$ to $\text{Fi}(A_i)$ for every rule $A_i \rightarrow w_i$
4. do steps 2 and 3 until all Fi sets stay the same.

Unfortunately, the First-sets are not sufficient to compute the parsing table. This is because a right-hand side w of a rule might ultimately be rewritten to the empty string. So the parser should also use the rule $A \rightarrow w$ if ϵ is in $\text{Fi}(w)$ and it sees on the input stream a symbol that could follow A . Therefore we also need the *Follow-set* of A , written as $\text{Fo}(A)$ here, which is defined as the set of terminals a such that there is a string of symbols $\alpha A a \beta$ that can be derived from the start symbol. Computing the Follow-sets for the nonterminals in a grammar can be done as follows:

1. initialize every $\text{Fo}(A_i)$ with the empty set
2. if there is a rule of the form $A_j \rightarrow w A_i w'$, then
 - if the terminal a is in $F_i(w')$, then add a to $\text{Fo}(A_i)$
 - if ϵ is in $F_i(w')$, then add $\text{Fo}(A_j)$ to $\text{Fo}(A_i)$
3. repeat step 2 until all Fo sets stay the same.

Now we can define exactly which rules will be contained where in the parsing table. If $T[A, a]$ denotes the entry in the table for nonterminal A and terminal a , then

$T[A, a]$ contains the rule $A \rightarrow w$ if and only if

- a is in $\text{Fi}(w)$ or
- ϵ is in $\text{Fi}(w)$ and a is in $\text{Fo}(A)$.

If the table contains at most one rule in every one of its cells, then the parser will always know which rule it has to use and can therefore parse strings without backtracking. It is in precisely this case that the grammar is called an *LL(1) grammar*.

Constructing an LL(k) parsing table

Until the mid 1990s, it was widely believed that LL(k) parsing (for $k > 1$) was impractical, since the parse table would have exponential size in k in the worst case. This perception changed gradually after the release of the PCCTS around 1992, when it was demonstrated that many programming languages can be parsed efficiently by an LL(k) parser without triggering the worst-case behavior of the parser. Moreover, in certain cases LL parsing is feasible even with unlimited lookahead. By contrast, traditional parser generators, like yacc use LALR(1) parse tables to construct a restricted LR parser with a fixed one-token lookahead.

Conflicts

As described in the introduction, LL(1) parsers recognize languages that have LL(1) grammars, which are a special case of context-free grammars (CFG's); LL(1) parsers cannot recognize all context-free languages. The LL(1) languages are exactly those recognized by deterministic pushdown automata restricted to a single state^[2]. In order for a CFG to be an LL(1) grammar, certain conflicts must not arise, which we describe in this section.

Terminology^[3]

Let A be a non-terminal. FIRST(A) is (defined to be) the set of terminals that can appear in the first position of any string derived from A. FOLLOW(A) is the union over FIRST(B) where B is any non-terminal that immediately follows A in the right hand side of a production rule.

LL(1) Conflicts

There are 3 types of LL(1) conflicts:

- FIRST/FIRST conflict
The FIRST sets of two different non-terminals intersect.
- FIRST/FOLLOW conflict
The FIRST and FOLLOW set of a grammar rule overlap. With an epsilon in the FIRST set it is unknown which alternative to select.

An example of an LL(1) conflict:

```
S -> A 'a' 'b'  
A -> 'a' | epsilon
```

The FIRST set of A now is { 'a' epsilon } and the FOLLOW set { 'a' }.

- left-recursion
Left recursion will cause a FIRST/FIRST conflict with all alternatives.

```
E -> E '+' term | alt1 | alt2
```

Solutions to LL(1) Conflicts

- Left-factoring

A common left-factor is factored out like $3x + 3y = 3(x+y)$.

```
A → X | X Y Z
```

becomes

```
A → X B
B → Y Z | ε
```

Can be applied when two alternatives start with the same symbol like a FIRST/FIRST conflict.

- Substitution

Substituting a rule into another rule to remove indirect or FIRST/FOLLOW conflicts. Note that this may cause a FIRST/FIRST conflict.

- Left recursion removal^[4]

A simple example for left recursion removal: The following production rule has left recursion on E

```
E → E '+' T
    -> T
```

This rule is nothing but list of T's separated by '+'. In a regular expression form $T ('+' T)^*$. So the rule could be rewritten as

```
E → T Z
Z → '+' T Z
    -> ε
```

Now there is no left recursion and no conflicts on either of the rules.

However, not all CFGs have an equivalent LL(k)-grammar, e.g.:

```
S → A | B
A → 'a' A 'b' | ε
B → 'a' B 'b' 'b' | ε
```

It can be shown that there does not exist any LL(k)-grammar accepting the language generated by this grammar.

External links

- An easy explanation of First and Follow Sets^[5] (an attempt to explain the process of creating first and follow sets in a more straightforward way)
- A tutorial on implementing LL(1) parsers in C#^[6]
- Parsing Simulator^[7] This simulator is used to generate parsing tables LL1 and to resolve the exercises of the book.
- LL(1) DSL PEG parser (toolkit framework)^[8]

Notes

- [1] <http://portal.acm.org/citation.cfm?id=805431>
- [2] Kurki-Suonio, R. (1969). "Notes on top-down languages". *BIT* 9 (3): 225–238. doi:10.1007/BF01946814.
- [3] <http://www.cs.uaf.edu/~cs331/notes/LL.pdf>
- [4] Modern Compiler Design, Grune, Bal, Jacobs and Langendoen
- [5] <http://www.jambe.co.nz/UNI/FirstAndFollowSets.html>
- [6] <http://www.itu.dk/people/kfl/parsernotes.pdf>
- [7] <http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php>
- [8] <http://expressionscompiler.codeplex.com/>

LR parser

In computer science, an **LR parser** is a parser that reads input from Left to right (as it would appear if visually displayed) and produces a **Rightmost derivation**. The term **LR(k) parser** is also used; where the k refers to the number of unconsumed "look ahead" input symbols that are used in making parsing decisions. Usually k is 1 and the term **LR parser** is often intended to refer to this case.

LR parsing can handle a larger range of languages than LL parsing, and is also better at error reporting, i.e. it detects syntactic errors when the input does not conform to the grammar as soon as possible. This is in contrast to an LL(k) (or even worse, an LL(*) parser) which may defer error detection to a different branch of the grammar due to backtracking, often making errors harder to localize across disjunctions with long common prefixes.

LR parsers are difficult to produce by hand and they are usually constructed by a parser generator or a compiler-compiler. Depending on how the parsing table is generated, these parsers can be called simple LR parsers (SLR), look-ahead LR parsers (LALR), or canonical LR parsers. LALR parsers have more language recognition power than SLR parsers. Canonical LR parsers have more recognition power than LALR parsers.

Theory

LR parsing was invented by Donald Knuth in 1965 in a paper, "On the Translation of Languages from Left to Right".

In this paper, Donald Knuth proves that "LR(k) grammars can be efficiently parsed with an execution time essentially proportional to the length of string", and that "A language can be generated by an LR(k) grammar if and only if it is deterministic, if and only if it can be generated by an LR(l) grammar" [1].

A deterministic context-free language is a language for which some LR(k) grammar exists. Every LR(k) grammar for $k > l$ can be mechanically transformed into an LR(l) grammar for the same language, while an LR(0) grammar for the same language may not exist; the LR(0) languages are a proper subset of the deterministic ones.

LR parsing can be generalized as arbitrary context-free language parsing without a performance penalty, even for LR(k) grammars. Note that the parsing of non-LR(k) context-free grammars is an order of magnitude slower (cubic instead of quadratic in relation to the input length).

The syntax of many programming languages are defined by a grammar that is LR(k) (where k is a small constant, usually 1), or close to being so, and for this reason LR parsers are often used by compilers to perform syntax analysis of source code.

An LR parser can be created from a context-free grammar by a program called a parser generator or hand written by a programmer. A context-free grammar is classified as LR(k) if there exists an LR(k) parser for it, as determined by the parser generator.

An LR parser is said to perform bottom-up parsing because it attempts to deduce the top level grammar productions by building up from the **leaves**.

An LR parser is based on an algorithm which is driven by a parser table, a data structure which contains the syntax of the computer language being parsed. So the term LR parser actually refers to a class of parser that can process almost any programming language, as long as the parser table for a programming language is supplied. The parser table is created by a program called a parser generator.

Architecture of LR parsers

Conceptually, an LR Parser is a recursive program that can be proven correct by direct computation, and can be implemented more efficiently (in time) as a recursive ascent parser, a set of mutually-recursive functions for every grammar, much like a recursive descent parser. Conventionally, however, LR parsers are presented and implemented as table-based stack machines in which the call stack of the underlying recursive program is explicitly manipulated.

A table-driven bottom-up parser can be schematically presented as in Figure 1. The following describes a rightmost derivation by this parser.

General case

The parser is a state machine. It consists of:

- an input buffer
- a stack on which is stored a list of states it has been in
- a goto table that prescribes to which new state it should move
- an action table that gives a grammar rule to apply given the current state and the current terminal in the input stream
- a set of CFL rules

Since the LR parser reads input from left to right but needs to produce a rightmost derivation, it uses reductions, instead of derivations to process input. That is, the algorithm works by creating a "leftmost reduction" of the input. The end result, when reversed, will be a rightmost derivation.

The LR parsing algorithm works as follows:

1. The stack is initialized with [0]. The current state will always be the state that is at the top of the stack.
2. Given the current state and the current terminal on the input stream an action is looked up in the action table.

There are four cases:

- a shift s_n :
 - the current terminal is removed from the input stream
 - the state n is pushed onto the stack and becomes the current state
 - a reduce r_m :
 - the number m is written to the output stream
 - for every symbol in the right-hand side of rule m a state is removed from the stack (i. e. if the right-hand side of rule m consists of 3 symbols, 3 top states are removed from the stack)
 - given the state that is then on top of the stack and the left-hand side of rule m a new state is looked up in the goto table and made the new current state by pushing it onto the stack
 - an accept: the string is accepted
 - no action: a syntax error is reported
3. Step 2 is repeated until either the string is accepted or a syntax error is reported.

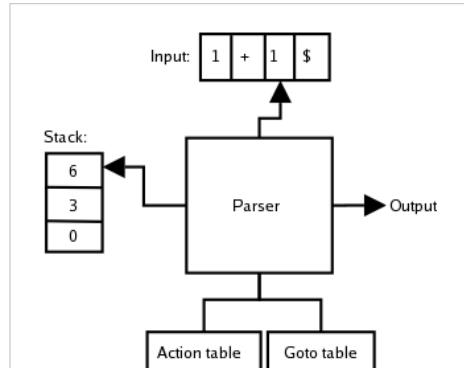


Figure 1. Architecture of a table-based bottom-up parser.

Concrete example

To explain its workings we will use the following small grammar whose start symbol is E:

- (1) $E \rightarrow E * B$
- (2) $E \rightarrow E + B$
- (3) $E \rightarrow B$
- (4) $B \rightarrow 0$
- (5) $B \rightarrow 1$

and parse the following input:

1 + 1

Action and goto tables

The two LR(0) parsing tables for this grammar look as follows:

state	action					goto	
	*	+	0	1	\$		
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

The **action table** is indexed by a state of the parser and a terminal (including a special terminal \$ that indicates the end of the input stream) and contains three types of actions:

- *shift*, which is written as 'sn' and indicates that the next state is *n*
- *reduce*, which is written as 'rm' and indicates that a reduction with grammar rule *m* should be performed
- *accept*, which is written as 'acc' and indicates that the parser accepts the string in the input stream.

The **goto table** is indexed by a state of the parser and a nonterminal and simply indicates what the next state of the parser will be if it has recognized a certain nonterminal. This table is important to find out the next state after every reduction. After a reduction, the next state is found by looking up the **goto table** entry for top of the stack (i.e. current state) and the reduced rule's LHS (i.e. non-terminal).

Parsing procedure

The table below illustrates each step in the process. Here the state refers to the element at the top of the stack (the right-most element), and the next action is determined by referring to the action table above. Also note that a \$ is appended to the input string to denote the end of the stream.

State	Input Stream	Output Stream	Stack	Next Action
0	1+1\$		[0]	Shift 2
2	+1\$		[0,2]	Reduce 5
4	+1\$	5	[0,4]	Reduce 3
3	+1\$	5,3	[0,3]	Shift 6
6	1\$	5,3	[0,3,6]	Shift 2
2	\$	5,3	[0,3,6,2]	Reduce 5
8	\$	5,3,5	[0,3,6,8]	Reduce 2
3	\$	5,3,5,2	[0,3]	Accept

Walkthrough

The parser starts out with the stack containing just the initial state ('0'):

[0]

The first symbol from the input string that the parser sees is '1'. In order to find out what the next action is (shift, reduce, accept or error), the action table is indexed with the current state (remember that the "current state" is just whatever is on the top of the stack), which in this case is 0, and the current input symbol, which is '1'. The action table specifies a shift to state 2, and so state 2 is pushed onto the stack (again, remember that all the state information is in the stack, so "shifting to state 2" is the same thing as pushing 2 onto the stack). The resulting stack is

[0 '1' 2]

where the top of the stack is 2. For the sake of explanation we also show the symbol (e.g., '1', B) that caused the transition to the next state, although strictly speaking it is not part of the stack.

In state 2 the action table says that regardless of what terminal we see on the input stream, we should do a reduction with grammar rule 5. If the table is correct, this means that the parser has just recognized the right-hand side of rule 5, which is indeed the case. In this case we write 5 to the output stream, pop one state from the stack (since the right-hand side of the rule has one symbol), and push on the stack the state from the cell in the goto table for state 0 and B, i.e., state 4. The resulting stack is:

[0 B 4]

However, in state 4 the action table says we should now do a reduction with rule 3. So we write 3 to the output stream, pop one state from the stack, and find the new state in the goto table for state 0 and E, which is state 3. The resulting stack:

[0 E 3]

The next terminal that the parser sees is a '+' and according to the action table it should then go to state 6:

[0 E 3 +' 6]

Note that the resulting stack can be interpreted as the history of a finite state automaton that has just read a nonterminal E followed by a terminal '+'. The transition table of this automaton is defined by the shift actions in the action table and the goto actions in the goto table.

The next terminal is now '1' and this means that we perform a shift and go to state 2:

[0 E 3 +' 6 '1' 2]

Just as the previous '1' this one is reduced to B giving the following stack:

[0 E 3 +' 6 B 8]

Again note that the stack corresponds with a list of states of a finite automaton that has read a nonterminal E, followed by a '+' and then a nonterminal B. In state 8 we always perform a reduce with rule 2. Note that the top 3 states on the stack correspond with the 3 symbols in the right-hand side of rule 2.

[0 E 3]

Finally, we read a '\$' from the input stream which means that according to the action table (the current state is 3) the parser accepts the input string. The rule numbers that will then have been written to the output stream will be [5, 3, 5, 2] which is indeed a rightmost derivation of the string "1 + 1" in reverse.

Constructing LR(0) parsing tables

Items

The construction of these parsing tables is based on the notion of *LR(0) items* (simply called *items* here) which are grammar rules with a special dot added somewhere in the right-hand side. For example the rule $E \rightarrow E + B$ has the following four corresponding items:

$$\begin{aligned} E &\rightarrow \bullet E + B \\ E &\rightarrow E \bullet + B \\ E &\rightarrow E + \bullet B \\ E &\rightarrow E + B \bullet \end{aligned}$$

Rules of the form $A \rightarrow \epsilon$ have only a single item $A \rightarrow \bullet$. The item $E \rightarrow E \bullet + B$, for example, indicates that the parser has recognized a string corresponding with E on the input stream and now expects to read a '+' followed by another string corresponding with B.

Item sets

It is usually not possible to characterize the state of the parser with a single item because it may not know in advance which rule it is going to use for reduction. For example if there is also a rule $E \rightarrow E * B$ then the items $E \rightarrow E \bullet + B$ and $E \rightarrow E \bullet * B$ will both apply after a string corresponding with E has been read. Therefore we will characterize the state of the parser by a set of items, in this case the set { $E \rightarrow E \bullet + B$, $E \rightarrow E \bullet * B$ }.

Extension of Item Set by expansion of non-terminals

An item with a dot before a nonterminal, such as $E \rightarrow E + \bullet B$, indicates that the parser expects to parse the nonterminal B next. To ensure the item set contains all possible rules the parser may be in the midst of parsing, it must include all items describing how B itself will be parsed. This means that if there are rules such as $B \rightarrow 1$ and $B \rightarrow 0$ then the item set must also include the items $B \rightarrow \bullet 1$ and $B \rightarrow \bullet 0$. In general this can be formulated as follows:

If there is an item of the form $A \rightarrow v \bullet Bw$ in an item set and in the grammar there is a rule of the form $B \rightarrow w'$ then the item $B \rightarrow \bullet w'$ should also be in the item set.

Closure of item sets

Thus, any set of items can be extended by recursively adding all the appropriate items until all nonterminals preceded by dots are accounted for. The minimal extension is called the *closure* of an item set and written as $\text{clos}(I)$ where I is an item set. It is these closed item sets that we will take as the states of the parser, although only the ones that are actually reachable from the begin state will be included in the tables.

Augmented grammar

Before we start determining the transitions between the different states, the grammar is always augmented with an extra rule

$$(0) S \rightarrow E$$

where S is a new start symbol and E the old start symbol. The parser will use this rule for reduction exactly when it has accepted the input string.

For our example we will take the same grammar as before and augment it:

$$(0) S \rightarrow E$$

$$(1) E \rightarrow E * B$$

$$(2) E \rightarrow E + B$$

$$(3) E \rightarrow B$$

$$(4) B \rightarrow 0$$

$$(5) B \rightarrow 1$$

It is for this augmented grammar that we will determine the item sets and the transitions between them.

Table construction

Finding the reachable item sets and the transitions between them

The first step of constructing the tables consists of determining the transitions between the closed item sets. These transitions will be determined as if we are considering a finite automaton that can read terminals as well as nonterminals. The begin state of this automaton is always the closure of the first item of the added rule: $S \rightarrow \bullet E$:

Item set 0

$$S \rightarrow \bullet E$$

$$+ E \rightarrow \bullet E * B$$

$$+ E \rightarrow \bullet E + B$$

$$+ E \rightarrow \bullet B$$

$$+ B \rightarrow \bullet 0$$

$$+ B \rightarrow \bullet 1$$

The boldfaced "+" in front of an item indicates the items that were added for the closure (not to be confused with the mathematical '+' operator which is a terminal). The original items without a "+" are called the *kernel* of the item set.

Starting at the begin state (S_0) we will now determine all the states that can be reached from this state. The possible transitions for an item set can be found by looking at the symbols (terminals and nonterminals) we find right after the dots; in the case of item set 0 those symbols are the terminals '0' and '1' and the nonterminals E and B . To find the item set that each symbol x leads to we follow the following procedure for each of the symbols:

1. Take the subset, S , of all items in the current item set where there is a dot in front of the symbol of interest, x .
2. For each item in S , move the dot to the right of x .

3. Close the resulting set of items.

For the terminal '0' (i.e. where $x = '0'$) this results in:

Item set 1

$$B \rightarrow 0 \cdot$$

and for the terminal '1' (i.e. where $x = '1'$) this results in:

Item set 2

$$B \rightarrow 1 \cdot$$

and for the nonterminal E (i.e. where $x = E$) this results in:

Item set 3

$$S \rightarrow E \cdot$$

$$E \rightarrow E \cdot * B$$

$$E \rightarrow E \cdot + B$$

and for the nonterminal B (i.e. where $x = B$) this results in:

Item set 4

$$E \rightarrow B \cdot$$

Note that the closure does not add new items in all cases - in the new sets above, for example, there are no nonterminals following the dot. We continue this process until no more new item sets are found. For the item sets 1, 2, and 4 there will be no transitions since the dot is not in front of any symbol. For item set 3 we see that the dot is in front of the terminals '*' and '+'. For '*' the transition goes to:

Item set 5

$$E \rightarrow E * \cdot B$$

$$+ B \rightarrow \cdot 0$$

$$+ B \rightarrow \cdot 1$$

and for '+' the transition goes to:

Item set 6

$$E \rightarrow E + \cdot B$$

$$+ B \rightarrow \cdot 0$$

$$+ B \rightarrow \cdot 1$$

For item set 5 we have to consider the terminals '0' and '1' and the nonterminal B. For the terminals we see that the resulting closed item sets are equal to the already found item sets 1 and 2, respectively. For the nonterminal B the transition goes to:

Item set 7

$$E \rightarrow E * B \cdot$$

For item set 6 we also have to consider the terminal '0' and '1' and the nonterminal B. As before, the resulting item sets for the terminals are equal to the already found item sets 1 and 2. For the nonterminal B the transition goes to:

Item set 8

$$E \rightarrow E + B \cdot$$

These final item sets have no symbols beyond their dots so no more new item sets are added and we are finished. The finite automaton, with item sets as its states is shown below.

The transition table for the automaton now looks as follows:

Item Set	*	+	0	1	E	B
0			1	2	3	4
1						
2						
3	5	6				
4						
5			1	2		7
6			1	2		8
7						
8						

Constructing the action and goto tables

From this table and the found item sets we construct the action and goto table as follows:

1. the columns for nonterminals are copied to the goto table
2. the columns for the terminals are copied to the action table as shift actions
3. an extra column for '\$' (end of input) is added to the action table that contains *acc* for every item set that contains $S \rightarrow E \bullet$.
4. if an item set i contains an item of the form $A \rightarrow w \bullet$ and $A \rightarrow w$ is rule m with $m > 0$ then the row for state i in the action table is completely filled with the reduce action rm .

The reader may verify that this results indeed in the action and goto table that were presented earlier on.

A note about LR(0) versus SLR and LALR parsing

Note that only step 4 of the above procedure produces reduce actions, and so all reduce actions must occupy an entire table row, causing the reduction to occur regardless of the next symbol in the input stream. This is why these are LR(0) parse tables: they don't do any lookahead (that is, they look ahead zero symbols) before deciding which reduction to perform. A grammar that needs lookahead to disambiguate reductions would require a parse table row containing different reduce actions in different columns, and the above procedure is not capable of creating such rows.

Refinements to the LR(0) table construction procedure (such as SLR and LALR) are capable of constructing reduce actions that do not occupy entire rows. Therefore, they are capable of parsing more grammars than LR(0) parsers.

Conflicts in the constructed tables

The automaton is constructed in such a way that it is guaranteed to be deterministic. However, when reduce actions are added to the action table it can happen that the same cell is filled with a reduce action and a shift action (a *shift-reduce conflict*) or with two different reduce actions (a *reduce-reduce conflict*). However, it can be shown that when this happens the grammar is not an LR(0) grammar.

A small example of a non-LR(0) grammar with a shift-reduce conflict is:

- (1) $E \rightarrow 1 E$
- (2) $E \rightarrow 1$

One of the item sets we then find is:

Item set 1

$E \rightarrow 1 \bullet E$

$$\begin{aligned} E &\rightarrow 1 \cdot \\ + E &\rightarrow \cdot 1 E \\ + E &\rightarrow \cdot 1 \end{aligned}$$

There is a shift-reduce conflict in this item set because in the cell in the action table for this item set and the terminal '1' there will be both a shift action to state 1 and a reduce action with rule 2.

A small example of a non-LR(0) grammar with a reduce-reduce conflict is:

$$\begin{aligned} (1) \quad E &\rightarrow A \ 1 \\ (2) \quad E &\rightarrow B \ 2 \\ (3) \quad A &\rightarrow 1 \\ (4) \quad B &\rightarrow 1 \end{aligned}$$

In this case we obtain the following item set:

Item set 1

$$\begin{aligned} A &\rightarrow 1 \cdot \\ B &\rightarrow 1 \cdot \end{aligned}$$

There is a reduce-reduce conflict in this item set because in the cells in the action table for this item set there will be both a reduce action for rule 3 and one for rule 4.

Both examples above can be solved by letting the parser use the follow set (see LL parser) of a nonterminal A to decide if it is going to use one of A 's rules for a reduction; it will only use the rule $A \rightarrow w$ for a reduction if the next symbol on the input stream is in the follow set of A . This solution results in so-called Simple LR parsers.

References

- *Compilers: Principles, Techniques, and Tools*, Aho, Sethi, Ullman, Addison-Wesley, 1986. ISBN 0-201-10088-6.
An extensive discussion of LR parsing and the principal source for some sections in this article.
- dickgrune.com^[2], Parsing Techniques - A Practical Guide 1st Ed. web page of book includes downloadable pdf.
- The Functional Treatment of Parsing (Boston: Kluwer Academic, 1993), R. Leermakers, ISBN 0-7923-9376-7
- The theory of parsing, translation, and compiling, Alfred V. Aho and Jeffrey D. Ullman, available from ACM.org
^[3]

[1] Knuth, Donald. [www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf] "On the Translation of Languages from Left to Right". www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf. Retrieved 29 May 2011.

[2] http://dickgrune.com/Books/PTAPG_1st_Edition/

[3] <http://portal.acm.org/citation.cfm?id=SERIES11430.578789>

Further reading

- Chapman, Nigel P., *LR Parsing: Theory and Practice* (<http://books.google.com/books?id=nEA9AAAAIAAJ&printsec=frontcover>), Cambridge University Press, 1987. ISBN 0-521-30413-X

External links

- Parsing Simulator (<http://www.supereeasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php>) This simulator is used to generate parsing tables LR and to resolve the exercises of the book
- Internals of an LALR(1) parser generated by GNU Bison (<http://www.cs.uic.edu/~spopuri/cparser.html>) - Implementation issues

Parsing table

This page is about LR parsing tables, there are also LL parsing tables which are substantially different.

A **parsing table** is the part of a parser that makes decisions about how to treat input tokens in compiler development.

Overview

A parsing table is a table describing what action its parser should take when a given input comes while it is in a given state. It is a tabular representation of a pushdown automaton that is generated from the context-free grammar of the language to be parsed. This is possible because the set of viable prefixes (that is, the sequence of input that can be set aside before a parser needs to either perform a reduction or return an error) of a context-free language is a regular language, so the parser can use a simple parse state table to recognize each viable prefix and determine what needs to be done next.

A parsing table is used with a stack and an input buffer. The stack starts out empty, and symbols are shifted onto it one by one from the input buffer with associated states; in each step, the parsing table is consulted to decide what action to take.

More simply, an Action table defines what to do when a terminal symbol is encountered, and a goto table defines what to do when a nonterminal is encountered.

The parsing table consists of two parts, the action table and the goto table. The action table takes the state at the top of the stack and the next symbol in the input buffer (called the "lookahead" symbol) and returns the action to take, and the next state to push onto the stack. The goto table returns the next state for the parser to enter when a reduction exposes a new state on the top of the stack. The goto table is needed to convert the operation of the deterministic finite automaton of the Action table to a pushdown automaton.

For example, a parsing table might take a current position in state 1 and an input of "+" from the input buffer, and return instructions to shift the current symbol onto the stack and move to state 4. More precisely, the parser would push the token "+" onto the stack, followed by the symbol 4. Or, for an example of the use of a goto table, the current stack might contain "E+E", which is supposed to reduce to "T". After that reduction, looking up "T" in the goto table row corresponding to the state currently at the top of the stack (that is, the state that was under the "E+E" popped off to do the reduction) would tell the parser to push state 2 on top of "T". Action may be a Reduction, a Shift, or Accept or None.

Reduction

Happens when the parser recognizes a sequence of symbols to represent a single nonterminal, and substitutes that nonterminal for the sequence. In an LR parser, this means that the parser will pop 2^*N entries from the stack (N being the length of the sequence recognized - this number is doubled because with each token pushed to the stack, a state is pushed on top, and removing the token requires that the parser also remove the state) and push the recognized nonterminal in its place. This nonterminal will not have an associated state, so to determine the next parse state, the parser will use the goto table.

This explanation can be made more precise using the "E+E" example above. Suppose the input buffer is currently empty and the contents of the stack are, from bottom to top:

0 E 1 + 5 E 1

The parser sees that there are no more tokens in the input stream and looks up the empty token in the parsing table. The table contains an instruction, for example, "r4" - this means to use reduction 4 to reduce the contents of the stack. Reduction 4, part of the grammar used to specify the parser's context-free language, should be something along the lines of:

T -> E+E

If the parser were to reduce the "E+E" to a "T", it would pop six symbols from the stack, leaving the stack containing:

0

and then push the new "T", followed by the state number found in the goto table at row 0, column T.

Shift

In this case, the parser makes the choice to shift the next symbol onto the stack. The token is moved from the input buffer to the stack along with an associated state, also specified in the action table entry, which will be the next parser state. The parser then advances to the next token.

Accept

When a parse is complete and the sequence of tokens in question can be matched to an accept state in the parse table, the parser terminates, and may report success in parsing the input. The input in question is valid for the selected grammar.

References

- "Lecture slides about generating parsing tables" ^[1], by Professor Alessandro Artale, researcher and lecturer at the Free University of Bolzano (from page 41).

References

[1] <http://www.inf.unibz.it/~artale/Compiler/slide4.pdf>

Simple LR parser

An SLR parser will typically have more conflict states than an LALR parser. For real-world computer languages, an SLR parser is usually not adequate, but for student projects in a compiler class it is a good learning tool.

A grammar that has no conflicts reported by an SLR parser generator is an *SLR grammar*.

Algorithm

The SLR parsing algorithm

```

Initialize the stack with S
Read input symbol
while (true)
    if Action(top(stack), input) = S
        NS <- Goto(top(stack), input)
        push NS
        Read next symbol
    else if Action(top(stack), input) = Rk
        output k
        pop |RHS| of production k from stack
        NS <- Goto(top(stack), LHS_k)
        push NS
    else if Action(top(stack), input) = A
        output valid, return
    else
        output invalid, return

```

Example

A grammar that can be parsed by an SLR parser but not by an LR(0) parser is the following:

- (0) $S \rightarrow E$
- (1) $E \rightarrow 1 E$
- (2) $E \rightarrow 1$

Constructing the action and goto table as is done for LR(0) parsers would give the following item sets and tables:

Item set 0

$S \rightarrow \bullet E$
 $+ E \rightarrow \bullet 1 E$
 $+ E \rightarrow \bullet 1$

Item set 1

$E \rightarrow 1 \bullet E$
 $E \rightarrow 1 \bullet$
 $+ E \rightarrow \bullet 1 E$
 $+ E \rightarrow \bullet 1$

Item set 2

$S \rightarrow E \bullet$

Item set 3

$$E \rightarrow 1 E \cdot$$

The action and goto tables:

	action		goto
state	1	\$	E
0	s1		2
1	s1/r2	r2	3
2		acc	
3	r1	r1	

As can be observed there is a shift-reduce conflict for state 1 and terminal '1'. However, the follow set of E is { \$ } so the reduce actions r1 and r2 are only valid in the column for \$. The result is the following conflict-less action and goto table:

	action		goto
state	1	\$	E
0	s1		2
1	s1	r2	3
2		acc	
3		r1	

Canonical LR parser

A **canonical LR parser** or **LR(1) parser** is an LR parser whose parsing tables are constructed in a similar way as with LR(0) parsers except that the items in the item sets also contain a *lookahead*, i.e., a terminal that is expected by the parser after the right-hand side of the rule. For example, such an item for a rule $A \rightarrow B C$ might be

$$A \rightarrow B \cdot C, a$$

which would mean that the parser has read a string corresponding to B and expects next a string corresponding to C followed by the terminal 'a'. LR(1) parsers can deal with a very large class of grammars but their parsing tables are often very big. This can often be solved by merging item sets if they are identical except for the lookahead, which results in so-called LALR parsers.

Constructing LR(1) parsing tables

An **LR(1) item** is a production with a marker together with a terminal, e.g., $[S \rightarrow a A \cdot B e, c]$. Intuitively, such an item indicates how much of a certain production we have seen already (a A), what we could expect next (B e), and a lookahead that agrees with what should follow in the input if we ever reduce by the production $S \rightarrow a A B e$. By incorporating such lookahead information into the item concept, we can make wiser reduce decisions. The lookahead of an LR(1) item is used directly only when considering reduce actions (i.e., when the \cdot marker is at the right end).

The **core** of an LR(1) item $[S \rightarrow a A \cdot B e, c]$ is the LR(0) item $S \rightarrow a A \cdot B e$. Different LR(1) items may share the same core. For example, if we have two LR(1) items of the form

- $[A \rightarrow \alpha \cdot, a]$ and
- $[B \rightarrow \alpha \cdot, b]$,

we take advantage of the lookahead to decide which reduction to use. (The same setting would perhaps produce a reduce/reduce conflict in the SLR approach.)

Validity

The notion of validity changes. An item $[A \rightarrow \beta_1 \cdot \beta_2, a]$ is **valid** for a viable prefix $\alpha \beta_1$ if there is a rightmost derivation that yields $\alpha A a w$ which in one step yields $\alpha \beta_1 \beta_2 a w$

Initial item

To get the parsing started, we begin with the initial item of

$[S' \rightarrow \cdot S, \$]$.

Here $\$$ is a special character denoting the end of the string.

Closure

Closure is more refined. If $[A \rightarrow \alpha \cdot B \beta, a]$ belongs to the set of items, and $B \rightarrow \gamma$ is a production of the grammar, then we add the item $[B \rightarrow \cdot \gamma, b]$ for all b in $\text{FIRST}(\beta a)$.

Goto

Goto is the same. A state containing $[A \rightarrow \alpha \cdot X \beta, a]$ will move to a state containing $[A \rightarrow \alpha X \cdot \beta, a]$ with label X . Every state has transitions according to Goto. for all

Shift actions

The shift actions are the same. If $[A \rightarrow \alpha \cdot b \beta, a]$ is in state I_k and I_k moves to state I_m with label b , then we add the action

$\text{action}[k, b] = \text{"shift } m\text{"}$

Reduce actions

The reduce actions are more refined than SLR . If $[A \rightarrow \alpha \cdot, a]$ is in state I_k , then we add the action: "Reduce $A \rightarrow \alpha$ " to $\text{action}[I_k, a]$. Observe that we don't use information from $\text{FOLLOW}(A)$ anymore. The goto part of the table is as before.

External links

- LR parsing ^[1] MS/Powerpoint presentation, Aggelos Kiayias, University of Connecticut

References

[1] <http://www.cse.uconn.edu/~akiayias/cse244fa04/N244-4-LR-more.ppt>

GLR parser

A **GLR parser** ("Generalized Left-to-right Rightmost derivation parser") is an extension of an LR parser algorithm to handle nondeterministic and ambiguous grammars. First described in a 1984 paper by Masaru Tomita, it has also been referred to as a "parallel parser". Tomita presented five stages in his original work^[1], though, in practice, it is the second stage that is recognized as the GLR parser.

Though the algorithm has evolved since its original form, the principles have remained intact: Tomita's goal was to parse natural language text thoroughly and efficiently. Standard LR parsers cannot accommodate the nondeterministic and ambiguous nature of natural language, and the GLR algorithm can.

Algorithm

Briefly, the GLR algorithm works in a manner similar to the LR parser algorithm, except that, given a particular grammar, a GLR parser will process all possible interpretations of a given input in a breadth-first search. On the front-end, a GLR parser generator converts an input grammar into parser tables, in a manner similar to an LR generator. However, where LR parse tables allow for only one state transition (given a state and an input token), GLR parse tables allow for multiple transitions. In effect, GLR allows for shift/reduce and reduce/reduce conflicts.

When a conflicting transition is encountered, the parse stack is forked into two or more parallel parse stacks, where the state corresponding to each possible transition is at the top. Then, the next input token is read and used to determine the next transition(s) for each of the "top" states – and further forking can occur. If any given top state and input token do not result in at least one transition, then that "path" through the parse tables is invalid and can be discarded.

A crucial optimization allows sharing of common prefixes and suffixes of these stacks, which constrains the overall search space and memory usage required to parse input text. The complex structures that arise from this improvement make the search graph a directed acyclic graph (with additional restrictions on the "depths" of various nodes), rather than a tree.

Advantages

The GLR algorithm has the same worst-case time complexity as the CYK algorithm and Earley algorithm – $O(n^3)$. However, GLR carries two additional advantages:

- The time required to run the algorithm is proportional to the degree of nondeterminism in the grammar – on deterministic grammars the GLR algorithm runs in $O(n)$ time (this is not true of the Earley and CYK algorithms, but the original Earley algorithms can be modified to ensure it)
- The GLR algorithm is "on-line" – that is, it consumes the input tokens in a specific order and performs as much work as possible after consuming each token.

In practice, the grammars of most programming languages are deterministic or "nearly deterministic," meaning that any nondeterminism is usually resolved within a small (though possibly unbounded) number of tokens. Compared to other algorithms capable of handling the full class of context-free grammars (such as Earley or CYK), the GLR algorithm gives better performance on these "nearly deterministic" grammars, because only a single stack will be active during the majority of the parsing process.

GLR can be combined with the LALR(1) algorithm, in a hybrid parser, allowing still higher performance^[2].

References

- [1] Masaru Tomita. Efficient parsing for natural language. Kluwer Academic Publishers, Boston, 1986.
- [2] <http://video.google.com/videoplay?docid=-8037498917324385899>

Further reading

- Grune, Dick; Jacobs, Ceriel J.H (2008). *Parsing Techniques*. Springer Science+Business Media. ISBN 978-0-387-20248-8.
- Tomita, Masaru (1984). "LR parsers for natural languages". *COLING*. 10th International Conference on Computational Linguistics. pp. 354–357.
- Tomita, Masaru (1985). "An efficient context-free parsing algorithm for natural languages". *IJCAI*. International Joint Conference on Artificial Intelligence. pp. 756–764.

LALR parser

An **LALR parser** is a type of parser defined in computer science as a Look-Ahead LR parser.

LALR parsers are based on a finite-state-automata concept. The data structure used by an LALR parser is a pushdown automaton (PDA). A deterministic PDA is a deterministic-finite automaton (DFA) with the addition of a stack for a memory, indicating which states the parser has passed through to arrive at the current state. Because of the stack, a PDA can recognize grammars that would be impossible with a DFA; for example, a PDA can determine whether an expression has any unmatched parentheses, whereas an automaton with no stack would require an infinite number of states due to unlimited nesting of parentheses.

LALR parsers are driven by a parser table in a finite-state machine (FSM) format. An FSM is tedious enough for humans to construct by hand that it is more convenient to use a software tool called an LALR parser generator to generate a parser table automatically from a grammar in Backus–Naur Form which defines the syntax of the computer language the parser will process^[1]. The original and most widely known parser generator was yacc, developed on the UNIX operating system. The parser table is often generated in source code format in a computer language (such as C++ or Java). When the parser (with parser table) is compiled and/or executed, it will recognize text files written in the language defined by the BNF grammar.

LALR parsers are generated from LALR grammars, which are capable of defining a larger class of languages than SLR grammars, but not as large a class as LR grammars. Real computer languages can often be expressed as LALR(1) grammars, and in cases where a LALR(1) grammar is insufficient, usually an LALR(2) grammar is adequate. If the parser generator handles only LALR(1) grammars, then the LALR parser will have to interface with some hand-written code when it encounters the special LALR(2) situation in the input language.

History

LR parsing was invented by Donald Knuth in 1965 in a paper, "On the Translation of Languages from Left to Right". LR parsers have the potential of providing the highest parsing speed of all parsing algorithms. However, LR parsers are considered impractical because of their huge size.

LALR parsing was invented by Frank DeRemer in 1969 in a paper, "Practical LR(k) Translators". LALR parsers offer the same high performance of LR parsers, but are much smaller in size. For this reason, it is LALR parsers that are most often generated by compiler-compilers such as yacc and GNU bison.

Generating LALR Parsers

Similar to an SLR parser generator, an LALR parser generator constructs the LR(0) state machine first and then computes the lookahead sets for all rules in the grammar, checking for ambiguity. An SLR parser generator computes the lookahead sets by examining the BNF grammar (these are called *follow sets*). However, an LALR parser generator computes the lookahead sets by examining the LR(0) state machine (these are called *LALR lookahead sets*, which are more accurate and less likely to cause a conflict/ambiguity to be reported by the parser generator).

Like SLR, LALR is a refinement to the technique for constructing LR(0) parse tables. While SLR uses *follow sets* to construct reduce actions, LALR uses *lookahead sets*, which are more specific because they take more of the parsing context into account. *Follow sets* are associated with a symbol, while *lookahead sets* are specific to an LR(0) item and a parser state.

Specifically, the *follow* set for a given LR(0) item I in a given parser state S contains all symbols that are allowed by the grammar to appear after I 's left-hand-side nonterminal. In contrast, the *lookahead* set for item I in state S contains only those symbols that are allowed by the grammar to appear after I 's right-hand-side has been parsed starting from state S . $\text{follow}(I)$ is effectively the union of the *lookahead* sets for all LR(0) items with the same left-hand-side as I , regardless of parser states or right-hand-sides, therefore losing all context information. Because the *lookahead* set is specific to a particular parsing context, it can be more selective, therefore allowing finer distinctions than the *follow* set.

Unfortunately, computing LALR lookahead sets is much more complicated than SLR. Frank DeRemer and Tom Pennello wrote a paper about this, published in SIGPLAN Notices in 1979 and in TOPLAS in 1982, called "Efficient Computation Of LALR(1) Look-Ahead Sets".

Advantages

1. An LALR parser can be automatically generated from an LALR grammar.
2. An LALR grammar can be used to define many computer languages.
3. An LALR parser is small.
4. An LALR parser is fast (if the parsing algorithm uses a matrix parser-table format).
5. An LALR parser is linear in speed (i.e. the speed is based on the size of the input text file only and not based on the size of the language being recognized).
6. The LALR grammar provides valuable documentation of the language being recognized.
7. Error recovery may already be built-in to the parser.
8. Generalized error messages may already be built into the parser.
9. Abstract-syntax-tree construction may already be built into the parser.
10. Recognition of context-sensitive language constructs may already be built into the parser.

Disadvantages

1. Software engineers are required to use an LALR parser generator, which may or may not be user friendly and may require some learning time.
2. Implementing meaningful error messages in the parser may be very difficult or impossible.
3. Understanding the parsing algorithm is often quite difficult.
4. If an error occurs, it may be difficult to determine whether it's in the grammar or the parser code.
5. If there is an error in the parser generator, this may be very difficult to fix.

References

- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* Addison-Wesley, 1986. (AKA The Dragon Book, describes the traditional techniques for building LALR(1) parsers.)
- Frank DeRemer and Thomas Pennello. Efficient Computation of LALR(1) Look-Ahead Sets^[2] *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4:4, pp. 615–649. 1982. (Describes a more efficient technique for building LALR(1) parsers.)
- Richard Bornat *Understanding and Writing Compilers*, Macmillan, 1979. (Describes the principles of automated left-to-right parsing and how to construct the parser tables, what a follow set is, etc., *in English, not mathematics* – available freely from the author's page at [3].)

External links

- Parsing Simulator^[7] This simulator is used to generate parsing tables LALR and resolve the exercises of the book.
- JS/CC^[4] JavaScript based implementation of a LALR(1) parser generator, which can be run in a web-browser or from the command-line.
- LALR(1) tutorial^[5] A flash card like tutorial on LALR(1) parsing.

[1] Levine, John; Mason, Tony; Brown, Doug. (1992). *Lex and Yacc..* O'Reilly Books., ISBN 1565920007.

[2] <http://portal.acm.org/citation.cfm?id=69622.357187>

[3] http://www.cs.mdx.ac.uk/staffpages/r_bornat/#vanitypublishing

[4] <http://jscc.jmksf.com/>

[5] <http://web.cs.dal.ca/~sjackson/lalr1.html>

Recursive ascent parser

In computing, **recursive ascent parsing** is a technique for implementing an LALR parser which uses mutually-recursive functions rather than tables. Thus, the parser is *directly encoded* in the host language similar to recursive descent. Direct encoding usually yields a parser which is faster than its table-driven equivalent^[1] for the same reason that compilation is faster than interpretation. It is also (nominally) possible to hand edit a recursive ascent parser, whereas a tabular implementation is nigh unreadable to the average human.

Recursive ascent was first described by Thomas Penello in his article "Very fast LR parsing"^[12] in 1986. He was not intending to create a hand-editable implementation of an LR parser, but rather a maintainable and efficient parser implemented in assembly language. The technique was later expounded upon by G.H. Roberts^[2] in 1988 as well as in an article by Leermakers, Augusteijn, Kruseman Aretz^[3] in 1992 in the journal *Theoretical Computer Science*. An extremely readable description of the technique was written by Morell and Middleton^[4] in 2003. A good exposition can also be found in a TOPLAS article by Sperber and Thiemann^[5].

Recursive ascent has also been merged with recursive descent, yielding a technique known as recursive ascent/descent. This implementation technique is arguably easier to hand-edit due to the reduction in states and fact that some of these states are more intuitively top-down rather than bottom up. It can also yield some minimal performance improvements over conventional recursive ascent^[6].

Summary

Intuitively, recursive ascent is a literal implementation of the LR parsing concept. Each function in the parser represents a single LR automaton state. Within each function, a multi-branch statement is used to select the appropriate action based on the current token popped off the input stack. Once the token has been identified, action is taken based on the state being encoded. There are two different fundamental actions which may be taken based on the token in question:

- **Shift** - Encoded as a function call, effectively jumping to a new automaton state.
- **Reduce** - Encoded differently according to the semantic action routine for the relevant production. The result of this routine is wrapped in an ADT which is returned to the caller. The reduce action must also record the number of tokens which were shifted *prior* to the reduce, passing this value back to the caller along with the reduce value. This shift counter determines at which point up the call stack the reduce should be handled.

There is also a third LR automaton action which may be taken in a given state, but only after a reduce where the shift counter has decremented to zero (indicating that the current state should handle the result). This is the **goto** action, which is essentially a special case of **shift** designed to handle non-terminals in a production. This action must be handled *after* the multi-branch statement, since this is where any reduction results will "resurface" from farther down the call stack.

Example

Consider the following grammar in bison syntax:

```
expr : expr '+' term    { $$ = $1 + $3; }
      | expr '-' term    { $$ = $1 - $3; }
      | term              { $$ = $1; }
      ;
term : '(' expr ')'   { $$ = $2; }
      | num               { $$ = $1; }
      ;
num : '0'                { $$ = 0; }
     | '1'                { $$ = 1; }
     ;
```

This grammar is LR(0) in that it is left-recursive (in the **expr** non-terminal) but does not require any lookahead. Recursive ascent is also capable of handling grammars which are LALR(1) in much the same way that table-driven parsers handle such cases (by pre-computing conflict resolutions based on possible lookahead).

The following is a Scala implementation of a recursive ascent parser based on the above grammar:

```
object ExprParser {
  private type Result = (NonTerminal, Int)

  private sealed trait NonTerminal {
    val v: Int
  }

  private case class NExpr(v: Int, in: Stream[Char]) extends NonTerminal
```

```

private case class NTterm(v: Int, in: Stream[Char]) extends
NonTerminal
private case class NTnum(v: Int, in: Stream[Char]) extends
NonTerminal

class ParseException(msg: String) extends RuntimeException(msg) {
  def this() = this("")

  def this(c: Char) = this(c.toString)
}

def parse(in: Stream[Char]) = state0(in)._1.v

/*
 * 0 $accept: . expr $end
 *
 * '(' shift, and go to state 1
 * '0' shift, and go to state 2
 * '1' shift, and go to state 3
 *
 * expr go to state 4
 * term go to state 5
 * num go to state 6
 */
private def state0(in: Stream[Char]) = in match {
  case cur #:: tail => {
    def loop(tuple: Result): Result = {
      val (res, goto) = tuple

      if (goto == 0) {
        loop(res match {
          case NTexpr(v, in) => state4(in, v)
          case NTterm(v, in) => state5(in, v)
          case NTnum(v, in) => state6(in, v)
        })
      } else (res, goto - 1)
    }

    loop(cur match {
      case '(' => state1(tail)
      case '0' => state2(tail)
      case '1' => state3(tail)
      case c => throw new ParseException(c)
    })
  }
}

case Stream() => throw new ParseException

```

```

}

/*
 * 4 term: '(' . expr ')'
 *
 * '(' shift, and go to state 1
 * '0' shift, and go to state 2
 * '1' shift, and go to state 3
 *
 * expr go to state 7
 * term go to state 5
 * num go to state 6
 */
private def state1(in: Stream[Char]): Result = in match {
  case cur #:: tail => {
    def loop(tuple: Result): Result = {
      val (res, goto) = tuple

      if (goto == 0) {
        loop(res match {
          case NTexpr(v, in) => state7(in, v)
          case NTterm(v, in) => state5(in, v)
          case NTnum(v, in) => state6(in, v)
        })
      } else (res, goto - 1)
    }

    loop(cur match {
      case '(' => state1(tail)
      case '0' => state2(tail)
      case '1' => state3(tail)
      case c => throw new ParseException(c)
    })
  }
}

case Stream() => throw new ParseException
}

/*
 * 6 num: '0' .
 *
 * $default reduce using rule 6 (num)
 */
private def state2(in: Stream[Char]) = (NTnum(0, in), 0)

/*
 * 7 num: '1' .

```

```
/*
 * $default reduce using rule 7 (num)
 */
private def state3(in: Stream[Char]) = (NTnum(1, in), 0)

/*
 * 0 $accept: expr . $end
 * 1 expr: expr . '+' term
 * 2      | expr . '-' term
 *
 * $end shift, and go to state 8
 * '+' shift, and go to state 9
 * '-' shift, and go to state 10
 */
private def state4(in: Stream[Char], arg1: Int): Result = in match {
  case cur #:: tail => {
    decrement(cur match {
      case '+' => state9(tail, arg1)
      case '-' => state10(tail, arg1)
      case c => throw new ParseException(c)
    })
  }
}

case Stream() => state8(arg1)
}

/*
 * 3 expr: term .
 *
 * $default reduce using rule 3 (expr)
 */
private def state5(in: Stream[Char], arg1: Int) = (NTexpr(arg1, in),
0)

/*
 * 5 term: num .
 *
 * $default reduce using rule 5 (term)
 */
private def state6(in: Stream[Char], arg1: Int) = (NTterm(arg1, in),
0)

/*
 * 1 expr: expr . '+' term
 * 2      | expr . '-' term
 * 4 term: '(' expr . ')'
 *
```

```

* '+'  shift, and go to state 9
* '-'  shift, and go to state 10
* ')'  shift, and go to state 11
*/
private def state7(in: Stream[Char], arg1: Int): Result = in match {
  case cur #:: tail => {
    decrement(cur match {
      case '+' => state9(tail, arg1)
      case '-' => state10(tail, arg1)
      case ')' => state11(tail, arg1)
      case c => throw new ParseException(c)
    })
  }
}

case Stream() => throw new ParseException
}

/*
* 0 $accept: expr $end .
*
* $default accept
*/
private def state8(arg1: Int) = (NTexpr(arg1, Stream()), 1)

/*
* 1 expr: expr '+' . term
*
* '(' shift, and go to state 1
* '0' shift, and go to state 2
* '1' shift, and go to state 3
*
* term go to state 12
* num  go to state 6
*/
private def state9(in: Stream[Char], arg1: Int) = in match {
  case cur #:: tail => {
    def loop(tuple: Result): Result = {
      val (res, goto) = tuple

      if (goto == 0) {
        loop(res match {
          case NTterm(v, in) => state12(in, arg1, v)
          case NTnum(v, in) => state6(in, v)
          case _ => throw new AssertionError
        })
      } else (res, goto - 1)
    }
  }
}

```

```
loop(cur match {
    case '(' => state1(tail)
    case '0' => state2(tail)
    case '1' => state3(tail)
    case c => throw new ParseException(c)
} )
}

case Stream() => throw new ParseException
}

/*
* 2 expr: expr '-' . term
*
* '(' shift, and go to state 1
* '0' shift, and go to state 2
* '1' shift, and go to state 3
*
* term go to state 13
* num go to state 6
*/
private def state10(in: Stream[Char], arg1: Int) = in match {
    case cur #:: tail => {
        def loop(tuple: Result): Result = {
            val (res, goto) = tuple

            if (goto == 0) {
                loop(res match {
                    case NTterm(v, in) => state13(in, arg1, v)
                    case NTnum(v, in) => state6(in, v)
                    case _ => throw new AssertionError
                })
            } else (res, goto - 1)
        }
    }

    loop(cur match {
        case '(' => state1(tail)
        case '0' => state2(tail)
        case '1' => state3(tail)
        case c => throw new ParseException(c)
    })
}

case Stream() => throw new ParseException
```

```

/*
 * 4 term: '(' expr ')'.
 *
 * $default reduce using rule 4 (term)
 */
private def state11(in: Stream[Char], arg1: Int) = (NTterm(arg1, in),
2)

/*
 * 1 expr: expr '+' term .
 *
 * $default reduce using rule 1 (expr)
 */
private def state12(in: Stream[Char], arg1: Int, arg2: Int) =
(NTexpr(arg1 + arg2, in), 2)

/*
 * 2 expr: expr '-' term .
 *
 * $default reduce using rule 2 (expr)
 */
private def state13(in: Stream[Char], arg1: Int, arg2: Int) =
(NTexpr(arg1 - arg2, in), 2)

private def decrement(tuple: Result) = {
  val (res, goto) = tuple
  assert(goto != 0)
  (res, goto - 1)
}
}

```

References

- [1] Thomas J Penello (1986). "Very fast LR parsing" (<http://portal.acm.org/citation.cfm?id=13310.13326>). .
- [2] G.H. Roberts (1988). "Recursive ascent: an LR analog to recursive descent" (<http://portal.acm.org/citation.cfm?id=47907.47909>). .
- [3] Leermakers, Augusteijn, Kruseman Aretz (1992). "A functional LR parser" (<http://portal.acm.org/citation.cfm?id=146986.146994>). .
- [4] Larry Morell and David Middleton (2003). "Recursive-ascent parsing" (<http://portal.acm.org/citation.cfm?id=770849>). .
- [5] Sperber and Thiemann (2000). "Generation of LR parsers by partial evaluation" (<http://portal.acm.org/citation.cfm?id=349219&dl=GUIDE&coll=GUIDE&CFID=56087236&CFTOKEN=74111863>). .
- [6] John Boyland and Daniel Spiewak (2009). "ScalaBison Recursive Ascent-Descent Parser Generator" (<http://www.cs.uwm.edu/~boyland/papers/scala-bison.pdf>). .

Parser combinator

In functional programming, the term ‘combinator’ is often used to denote functions which combine a number of things to a new thing of the same kind. When combinators are used to construct a parsing technique, then they are called **parser combinators** and the parsing method is called **combinatory-parsing** (as higher-order functions ‘combine’ different parsers together). Parser combinators use a top-down parsing strategy which facilitates modular piecewise construction and testing.

Parsers built using combinators are straightforward to construct, ‘readable’, modular, well-structured and easily maintainable. They have been used extensively in the prototyping of compilers and processors for domain-specific languages such as natural language interfaces to databases, where complex and varied semantic actions are closely integrated with syntactic processing. In 1989, Richard Frost and John Launchbury demonstrated^[1] use of parser combinators to construct Natural Language interpreters. Graham Hutton also used higher-order functions for basic parsing in 1992.^[2] S.D. Swierstra also exhibited the practical aspects of parser combinators in 2001.^[3] In 2008, Frost, Hafiz and Callaghan described^[4] a set of parser-combinators in Haskell that solve the long standing problem of accommodating left-recursion, and work as a complete top-down parsing tool in polynomial time and space.

Basic idea

In functional programming, parser combinators can be used to build basic parsers and to construct complex parsers for rules (that define nonterminals) from other parsers. A production-rule of a context-free grammar (CFG) may have one or more ‘alternatives’ and each alternative may consist of a sequence of non-terminal(s) and/or terminal(s), or the alternative may consist of a single non-terminal or terminal or ‘empty’. Parser combinators allow parsers to be defined in an embedded style, in code which is similar in structure to the rules of the grammar. As such, implementations can be thought of as executable specifications with all of the associated advantages. In order to achieve this, one has to define a set of combinators or infix operators to ‘glue’ different terminals and non-terminals to form a complete rule.

The combinators

To keep the discussion relatively straight forward, we discuss parser combinators in terms of recognizer only. Assume that the input is a length `#input` sequence of tokens, the members of which are accessed through an index `j`. Recognizers are functions which take an index `j` as argument and which return a set of indices. Each index in the result set corresponds to a position at which the parser successfully finished recognizing a sequence of tokens that began at position `j`. An empty result set indicates that the recognizer failed to recognize any sequence beginning at `j`. A non-empty result set indicates the recognizer ends at different positions successfully. (Note that, as we are defining results as a set, we cannot express ambiguity as it would require repeated entries in the set. Use of ‘list’ would solve the problem.) Following the definitions of two basic recognizers for terminals, we define two major combinators for alternative and sequencing:

- The `empty` recognizer is a function which always succeeds returning a singleton set containing the current position:

$$\text{empty}(j) = \{j\}$$

- A recognizer `term 'x'` for a terminal `x` is a function which takes an index `j` as input, and if `j` is less than `#input` and if the token at position `j` in the input corresponds to the terminal `x`, it returns a singleton set containing `j + 1`, otherwise it returns the empty set.

$$term(x, j) = \begin{cases} \{\}, & j \geq \#input \\ \{j + 1\}, & j^{th} \text{ element of } input = x \\ \{\}, & \text{otherwise} \end{cases}$$

- We call the ‘alternative’ combinator $\langle + \rangle$, which is used as an infix operator between two recognizers p and q . The $\langle + \rangle$ applies both of the recognizers on the same input position j and sums up the results returned by both of the recognizers, which is eventually returned as the final result.

$$(p \quad \langle + \rangle \quad q)(j) = p(j) \cup q(j)$$

- The sequencing of recognizers is done with the $\langle * \rangle$ combinator. Like $\langle + \rangle$, it is also used as an infix operator between two recognizers – p and q . But it applies the first recognizer p to the input position j , and if there is any successful result of this application, then the second recognizer q is applied to every element of the result set returned by the first recognizer. The $\langle * \rangle$ ultimately returns the union of these applications of q .

$$(p \quad \langle * \rangle \quad q)(j) = \bigcup \{q(k) : k \in p(j)\}$$

Examples

- Consider a highly ambiguous CFG $s ::= 'x' s \mid s \mid \epsilon$. Using the combinators defined earlier, we can modularly define executable notations of this grammar in a modern functional language (e.g. Haskell) as $s = \text{term} 'x' \langle * \rangle s \langle * \rangle s \langle + \rangle \text{empty}$. When the recognizer s is applied on an input sequence $xxxx$ at position 1, according to the above definitions it would return a result set $\{5, 4, 3, 2, 1\}$. Note that in real implementation if result set is defined as data type that supports repetition (i.e. list), then we can have the resulting list with all possible ambiguous results like $[5, 4, 3, 2, 1, \dots, 5, 4, 3, 2, \dots]$.

Shortcomings and solutions

Parser combinators, like all recursive descent parsers, are not limited to the Context-free grammar and thus do no global search for ambiguities in the LL(k) parsing First_k and Follow_k sets. Thus ambiguities are not known until run-time if and until the input triggers them. Such ambiguities where the recursive descent parser defaults (perhaps unknown to the grammar designer) to one of the possible ambiguous paths, thus results in semantic confusion (aliasing) in the use of the language, and leads to bugs by users of ambiguous programming languages which are not reported at compile-time, and which are introduced not by human error, but by ambiguous grammar. The only solution which eliminates these bugs, is to remove the ambiguities and use a Context-free grammar.

The simple implementations of parser combinators have some shortcomings, which are common in top-down parsing. Naïve combinatory parsing requires exponential time and space when parsing an ambiguous context free grammar. In 1996, Frost and Szydłowski^[5] demonstrated how memoization can be used with parser combinators to reduce the time complexity to polynomial. Later Frost used monads^[6] to construct the combinators for systematic and correct threading of memo-table throughout the computation.

Like any top-down recursive descent parsing, the conventional parser combinators (like the combinators described above) won’t terminate while processing a left-recursive grammar (i.e. $s ::= s \langle * \rangle s \langle * \rangle \text{term} 'x' \mid \text{empty}$). A recognition algorithm^[7] that accommodates ambiguous grammars with direct left-recursive rules is described by Frost and Hafiz in 2006. The algorithm curtails the otherwise ever-growing left-recursive parse by imposing depth restrictions. That algorithm was extended^[8] to a complete parsing algorithm to accommodate indirect as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially-exponential number of parse trees for highly-ambiguous grammars by Frost, Hafiz and Callaghan in 2007. This extended algorithm accommodates indirect left-recursion by comparing its ‘computed-context’ with ‘current-context’. The same authors also described^[4] their implementation of a set of parser combinators written in the Haskell programming language based on the same algorithm. The X-SAIGA^[7] site has more about the

algorithms and implementation details.

References

- [1] Frost, Richard. and Launchbury, John. "Constructing natural language interpreters in a lazy functional language." *The Computer Journal — Special edition on Lazy Functional Programming* Volume 32, Issue 2. Pages: 108 – 121, 1989
- [2] Hutton, Graham. "Higher-order functions for parsing." *Journal of Functional Programming*, Volume 2 Issue 3, Pages: 323– 343, 1992.
- [3] Swierstra, S. Doaitse. "Combinator parsers: From toys to tools. " In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.
- [4] Frost, R., Hafiz, R. and Callaghan, P." Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902, year 2008, Pages: 167-181, January 2008, San Francisco.
- [5] Frost, Richard. and Szydłowski, Barbara. "Memoizing Purely Functional Top-Down Backtracking Language Processors. " *Sci. Comput. Program.* 1996 - 27(3): 263-288.
- [6] Frost, Richard. "Monadic Memoization towards Correctness-Preserving Reduction of Search. " *Canadian Conference on AI 2003.* p 66-80.
- [7] Frost, R. and Hafiz, R." A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time." *ACM SIGPLAN Notices*, Volume 41 Issue 5, Pages: 46 - 54. Year: 2006
- [8] Frost, R., Hafiz, R. and Callaghan, P." Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109 - 120, June 2007, Prague.

External links

- X-SAIGA (<http://www.cs.uwindsor.ca/~hafiz/proHome.html>) - eXecutable SpecificAtIons of GrAmmars

Bottom-up parsing

Bottom-up parsing is a strategy for analyzing unknown information that attempts to identify the most fundamental units first, and then to infer higher-order structures from them. The name comes from the concept of a parse tree, in which the most fundamental units are at the bottom, and structures composed of them are in successively higher layers, until at the top of the tree a single unit, or *start symbol*, comprises all of the information being analyzed.

Bottom-up parsing occurs in the analysis of both natural languages and computer languages. It is contrasted with top-down parsing, in which complex units are divided into simpler parts until all of the information is parsed.

Linguistics

In **linguistics**, an example of bottom-up parsing would be analyzing a sentence by identifying words first, and then using properties of the words to infer grammatical relations and phrase structures to build a parse tree of the complete sentence.

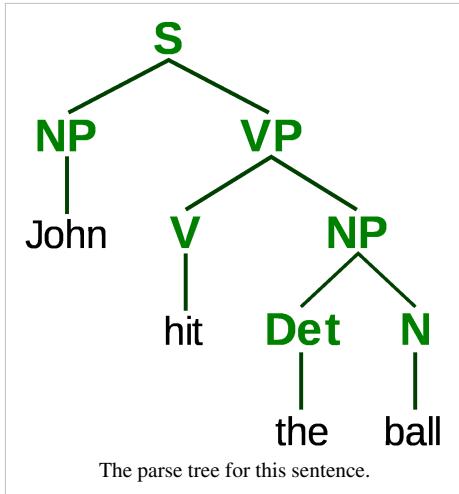
Example

The English sentence "John hit the ball" may be analysed by bottom-up parsing. In this case, the most fundamental units (ignoring conjugation and declension) are the words: "John", "hit", "the" and "ball".

A simplified grammar for English, sufficient for this sentence, is:

1. A sentence can be a noun phrase followed by a verb phrase ($S \rightarrow NP\ VP$)
2. A noun phrase can be a noun by itself ($NP \rightarrow N$)
3. A noun phrase can also be a determiner followed by a noun ($NP \rightarrow Det\ N$)
4. A verb phrase can be a verb followed by a noun phrase ($VP \rightarrow V\ NP$)
5. Nouns, determiners and verbs are all words ($N \rightarrow \text{word}$, $Det \rightarrow \text{word}$, $V \rightarrow \text{word}$)

Proceeding bottom-up from these words, we find:



Rule used	Words parsed	Sentence now
5	In this case, "John" and "ball" are nouns, "the" (the definite article) is a determiner, and "hit" is a verb.	N V Det N
3	"the ball" is a noun phrase.	N V NP
4	"hit the ball" is a verb phrase.	N VP
2	"John" is a noun phrase by itself.	NP VP
1	"John hit the ball" is a complete sentence.	S

Other parsings are possible; for example, "ball" could have been parsed as a noun phrase on its own. However, no other parsing can lead to a complete sentence using these simplified rules, as once "ball" is chosen to be a noun phrase, there is no rule that lets us parse "the". A brute-force search can be used to find all possible parsings of the sentence; if more than one parsing is possible, the sentence is ambiguous -- not uncommon in natural languages.

Computer science

In **programming language** compilers, bottom-up parsing is a parsing method that works by identifying terminal symbols first, and combines them successively to produce nonterminals. The productions of the parser can be used to build a parse tree of a program written in human-readable source code that can be compiled to assembly language or pseudocode.

Different computer languages require different parsing techniques, although it is not uncommon to use a parsing technique that is more powerful than that actually required.

It is common for bottom-up parsers to take the form of general parsing engines, which can either parse, or generate a parser for, a specific programming language given a specification of its grammar. Perhaps the most well known generalized parser generators are YACC and GNU bison.

Example

This trivial grammar defines simple addition expressions:

1. A complete expression (a sum) can be an addend, the plus sign, and another addend ($S \rightarrow A \ " + " \ A$)
2. An addend can be a letter (a variable) ($A \rightarrow \text{letter}$)
3. An addend can also be a number ($A \rightarrow N$)
4. An addend can also be a sum itself ($A \rightarrow S$)
5. A number is one or more digits ($N \rightarrow \text{digit}^+$)

(The plus sign in rule 5 signifies repetition: "one or more digits". To distinguish it from this special use, the plus sign in rule 1 is placed inside quotation marks.)

The fundamental units of this grammar are single letters, the plus sign, and groups of one or more digits. So, in the input "a + x + 12", the fundamental units are "a", "+", "x", "+", and "12". A parsing strategy might look like this:

Rule used	Symbols parsed	Expression now
5	"12" is a number.	a + x + N
2, 3	"a", "x" and "12" are addends.	A + A + A
1	"a + x" is a complete sum.	S + A
4	The sum "a + x" is also an addend.	A + A
1	"a + x + 12" is a complete sum.	S

Note that even though a complete expression was found in the third step, there was still some input remaining, so the parsing had to continue.

As with the linguistics example above, other parsings are possible; specifically, at the third step, "x + 12" could also have been legitimately parsed as a sum. This would have also resulted in a successfully completed parsing; thus, the grammar given is ambiguous. Unlike natural languages, computer languages cannot be ambiguous, so bottom-up parsing for computer languages must have additional rules, such as "the leftmost possible replacement is the one chosen". Additionally, a computer language parser will rarely examine the entire input at once, since the input may be a very large computer program; almost all parsers read the input from start to end (usually called "left to right"). However, the ability to check one or more upcoming symbols ("lookahead") before applying a rule increases the expressive power of a grammar.

Type of bottom-up parsers

The common classes of bottom-up parsers are:

- LR parser
 - LR(0) - No lookahead symbol
 - SLR(1) - Simple with one lookahead symbol
 - LALR(1) - Lookahead bottom up, not as powerful as full LR(1) but simpler to implement. YACC deals with this kind of grammar.
 - LR(1) - Most general grammar, but most complex to implement.
 - LR(n) - (where n is a positive integer) indicates an LR parser with n lookahead symbols; while grammars can be designed that require more than 1 lookahead, practical grammars try to avoid this because increasing n can theoretically require exponentially more code and data space (in practice, this may not be as bad). Also, the class of LR(n) languages is the same as that of LR(1) languages.
- Precedence parsers
 - Simple precedence parser

- Operator-precedence parser
- Extended precedence parser

Bottom-up parsing can also be done by backtracking.

Shift-reduce parsers

The most common bottom-up parsers are the shift-reduce parsers. These parsers examine the input tokens and either shift (push) them onto a stack or reduce elements at the top of the stack, replacing a right-hand side by a left-hand side.

Action table

Often an action (or parse) table is constructed which helps the parser determine what to do next. The following is a description of what can be held in an action table.

Actions

- Shift - push token onto stack
- Reduce - remove handle from stack and push on corresponding nonterminal
- Accept - recognize sentence when stack contains only the distinguished symbol and input is empty
- Error - happens when none of the above is possible; means original input was not a sentence

Shift and reduce

A shift-reduce parser uses a stack to hold the grammar symbols while awaiting reduction. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input.

The parser is a stack automaton which is in one of several discrete states. In reality, in the case of LR parsing, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

Algorithm: Shift-reduce parsing

1. Start with the sentence to be parsed as the initial sentential form
2. Until the sentential form is the start symbol do:
 1. Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (this is called a handle)
 2. Apply a production rule in reverse; i.e., replace the RHS of the rule which appears in the sentential form with the LHS of the rule (an action known as a reduction)

In step 2.1 above we are "shifting" the input symbols to one side as we move through them; hence a parser which operates by repeatedly applying steps 2.1 and 2.2 above is known as a shift-reduce parser.

A shift-reduce parser is most commonly implemented using a stack, where we proceed as follows:

- start with an empty stack
- a "shift" action corresponds to pushing the current input symbol onto the stack
- a "reduce" action occurs when we have a handle on top of the stack. To perform the reduction, we pop the handle off the stack and replace it with the nonterminal on the LHS of the corresponding rule.

Example

Take the grammar:

```

Sentence --> NounPhrase VerbPhrase
NounPhrase --> Art Noun
VerbPhrase --> Verb | Adverb Verb
Art --> the | a | ...
Verb --> jumps | sings | ...
Noun --> dog | cat | ...
  
```

And the input:

the dog jumps

Then the bottom up parsing is:

Stack	Input Sequence	
()	(the dog jumps)	
(the)	(dog jumps)	SHIFT word onto stack
(Art)	(dog jumps)	REDUCE using grammar rule
(Art dog)	(jumps)	SHIFT..
(Art Noun)	(jumps)	REDUCE..
(NounPhrase)	(jumps)	REDUCE
(NounPhrase jumps)	()	SHIFT
(NounPhrase Verb)	()	REDUCE
(NounPhrase VerbPhrase)()		REDUCE
(Sentence)		SUCCESS

Another Example

Given the grammar:

```

<Expression> --> <Term> | <Term> + <Expression>
<Term> --> <Factor> | <Factor> * <Term>
<Factor> --> [ <Expression> ] | 0...9
  
```

Stack	Input String	Action
=====	=====	=====
()	(2 * [1 + 3])	SHIFT
(2)	(* [1 + 3])	REDUCE
(<Factor>)	(* [1 + 3])	SHIFT
(<Factor> *)	([1 + 3])	SHIFT
(<Factor> * [)	(1 + 3])	SHIFT
(<Factor> * [1)	(+ 3])	REDUCE
(<Factor> * [<Factor>)	(+ 3])	REDUCE
(<Factor> * [<Term>)	(+ 3])	SHIFT
(<Factor> * [<Term> +)	(3])	SHIFT
(<Factor> * [<Term> + 3)	(])	REDUCE
(<Factor> * [<Term> + <Factor>)	(])	REDUCE
(<Factor> * [<Term> + <Term>)	(])	REDUCE
(<Factor> * [<Term> + <Expression>)	(])	REDUCE

(<Factor> * [<Expression>)	(])	SHIFT
(<Factor> * [<Expression>])	()	REDUCE
(<Factor> * <Factor>)	()	REDUCE
(<Factor> * <Term>)	()	REDUCE
(<Term>)	()	REDUCE
(<Expression>)	()	SUCCESS

External links

- Parsing Simulator ^[7] This simulator is used to generate examples of shift-reduce parsing
- An example of shift-reduce parsing ^[1] (which is a type of bottom up parsing), with a small grammar, state diagram, and C language code to implement the parser
- Course notes on shift reduce parsing ^[2]
- A good non-technical tutorial in the context of natural (human) languages ^[3] (Instead use an archived version of the page ^[4])
- A discussion of shift-reduce conflicts in bottom up parsers ^[5]. A knowledgeable but technical article.
- Yet another bottom-up parsing illustration ^[6]

References

- [1] <http://lambda.uta.edu/cse5317/notes/node18.html>
[2] <http://www.cs.grinnell.edu/~rebelsky/Courses/CS362/2004S/Outlines/outline.20.html>
[3] <http://nltk.sourceforge.net/tutorial/parsing/section-approaches.html>
[4] <http://web.archive.org/web/20070313000407/http://nltk.sourceforge.net/tutorial/parsing/section-approaches.html>
[5] <http://www.gobosoft.com/eiffel/gobo/geyacc/algorithm.html>
[6] <http://www.cs.uky.edu/~lewis/essays/compilers/bu-parse.html>

Chomsky normal form

In computer science, a context-free grammar is said to be in **Chomsky normal form** if all of its production rules are of the form:

$$A \rightarrow BC \text{ or}$$

$$A \rightarrow \alpha \text{ or}$$

$$S \rightarrow \epsilon$$

where A , B and C are nonterminal symbols, α is a terminal symbol (a symbol that represents a constant value), S is the start symbol, and ϵ is the empty string. Also, neither B nor C may be the start symbol.

Every grammar in Chomsky normal form is context-free, and conversely, every context-free grammar can be transformed into an equivalent one which is in Chomsky normal form. Several algorithms for performing such a transformation are known. Transformations are described in most textbooks on automata theory, such as (Hopcroft and Ullman, 1979). As pointed out by Lange and Leiß, the drawback of these transformations is that they can lead to an undesirable bloat in grammar size. Using $|G|$ to denote the size of the original grammar G , the size blow-up in the worst case may range from $|G|^2$ to $2^{2|G|}$, depending on the transformation algorithm used (Lange and Leiß, 2009).

Alternative definition

Another way to define Chomsky normal form (e.g., Hopcroft and Ullman 1979, and Hopcroft et al. 2006) is:

A formal grammar is in **Chomsky reduced form** if all of its production rules are of the form:

$$A \rightarrow BC \text{ or}$$

$$A \rightarrow \alpha$$

where A , B and C are nonterminal symbols, and α is a terminal symbol. When using this definition, B or C may be the start symbol.

Converting a grammar to Chomsky Normal Form

1. Introduce S_0

Introduce a new start variable, S_0 and a new rule $S_0 \rightarrow S$ where S is the previous start variable.

2. Eliminate all ϵ rules

ϵ rules are rules of the form $A \rightarrow \epsilon$ where $A \neq S_0$ and $A \in V$ where V is the CFG's variable alphabet.

Remove every rule with ϵ on its right hand side (RHS). For each rule with A in its RHS, add a set of new rules consisting of the different possible combinations of A replaced or not replaced with ϵ . If a rule has A as a singleton on its RHS, add a new rule $R = A \rightarrow \epsilon$ unless R has already been removed through this process. For example, examine the following grammar G:

$$S \rightarrow AbA|B$$

$$B \rightarrow b|c$$

$$A \rightarrow \epsilon$$

G has one epsilon rule. When the $A \rightarrow \epsilon$ is removed, we get the following:

$$S \rightarrow AbA|Ab|bA|b|B$$

$$B \rightarrow b|c$$

Notice that we have to account for all possibilities of $A \rightarrow \epsilon$ and so we actually end up adding 3 rules.

3. Eliminate all unit rules

$$A \rightarrow B \ni A, B \in V$$

After all the ϵ rules have been removed, you can begin removing unit rules, or rules whose RHS contains one variable and no terminals (which is inconsistent with CNF.)

To remove $A \rightarrow B$

$\forall B \rightarrow U$ add rule $A \rightarrow U$ unless this is a unit rule which has already been removed.

4. Clean up remaining rules that are not in Chomsky normal form.

Replace

$A \rightarrow u_1, u_2, \dots, u_k, k \geq 3, u_i \in V \cup \Sigma$ with

$A \rightarrow u_1 A_1, A_1 \rightarrow u_2 A_2, \dots, A_{k-2} \rightarrow u_{k-1} u_k$ where A_i are new variables.

If $u_i \in \Sigma$, replace u_i in above rules with some new variable v_i and add rule $v_i \rightarrow u_i$.

References

- John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd Edition, Addison-Wesley, 2006. ISBN 0-321-45536-3. (See subsection 7.1.5, page 272.)
- John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing, Reading Massachusetts, 1979. ISBN 0-201-02988-X. (See chapter 4.)
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. (Pages 98–101 of section 2.1: context-free grammars. Page 156.)
- John Martin (2003). *Introduction to Languages and the Theory of Computation*. McGraw Hill. ISBN 0-07-232200-4. (Pages 237–240 of section 6.6: simplified forms and normal forms.)
- Michael A. Harrison (1978). *Introduction to Formal Language Theory*. Addison-Wesley. ISBN 978-0201029550. (Pages 103–106.)
- Lange, Martin and Leiß, Hans. *To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm*. Informatica Didactica 8, 2009. ((pdf) ^[1])
- Cole, Richard. *Converting CFGs to CNF (Chomsky Normal Form)*, October 17, 2007. (pdf) ^[2]
- Sipser, Michael. *Introduction to the Theory of Computation*, 2nd edition.

References

[1] <http://ddi.cs.uni-potsdam.de/InformaticaDidactica/LangeLeiss2009.pdf>

[2] <http://cs.nyu.edu/courses/fall07/V22.0453-001/cnf.pdf>

CYK algorithm

The **Cocke–Younger–Kasami (CYK) algorithm** (alternatively called **CKY**) is a parsing algorithm for context-free grammars. It employs bottom-up parsing and dynamic programming.

The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language (Sipser 1997).

The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Landau symbols, the worst case running time of CYK is $\Theta(n^3 \cdot |G|)$, where n is the length of the parsed string and $|G|$ is the size of the CNF grammar G . This makes it the most efficient parsing algorithm in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

Standard form

The algorithm requires the context-free grammar to be rendered into Chomsky normal form (CNF), because it tests for possibilities to split the current sequence in half. Any context-free grammar can be represented in CNF using only rules of the forms $A \rightarrow \alpha$ and $A \rightarrow BC$.

As pseudocode

The algorithm in pseudocode is as follows:

```

let the input be a string  $S$  consisting of  $n$  characters:  $a_1 \dots a_n$ .
let the grammar contain  $r$  nonterminal symbols  $R_1 \dots R_r$ .
This grammar contains the subset  $R_s$  which is the set of start symbols.
let  $P[n, n, r]$  be an array of booleans. Initialize all elements of  $P$  to false.
for each  $i = 1$  to  $n$ 
  for each unit production  $R_j \rightarrow a_i$ 
    set  $P[i, 1, j] = \text{true}$ 
for each  $i = 2$  to  $n$  -- Length of span
  for each  $j = 1$  to  $n-i+1$  -- Start of span
    for each  $k = 1$  to  $i-1$  -- Partition of span
      for each production  $R_A \rightarrow R_B R_C$ 
        if  $P[j, k, B]$  and  $P[j+k, i-k, C]$  then set  $P[j, i, A] = \text{true}$ 
if any of  $P[1, n, x]$  is true ( $x$  is iterated over the set  $s$ , where  $s$  are all the indices for  $R_s$ ) then
   $S$  is member of language
else
   $S$  is not member of language

```

As prose

In informal terms, this algorithm considers every possible subsequence of the sequence of words and sets $P[i,j,k]$ to be true if the subsequence of words starting from i of length j can be generated from R_k . Once it has considered subsequences of length 1, it goes on to subsequences of length 2, and so on. For subsequences of length 2 and greater, it considers every possible partition of the subsequence into two parts, and checks to see if there is some production $P \rightarrow Q R$ such that Q matches the first part and R matches the second part. If so, it records P as matching the whole subsequence. Once this process is completed, the sentence is recognized by the grammar if the subsequence containing the entire sentence is matched by the start symbol.

CYK table

S						
	VP					
S						
	VP			PP		
S		NP			NP	
NP	V, VP	Det.	N	P	Det	N
she	eats	a	fish	with	a	fork

Extensions

Generating a parse tree

It is simple to extend the above algorithm to not only determine if a sentence is in a language, but to also construct a parse tree, by storing parse tree nodes as elements of the array, instead of booleans. Since the grammars being recognized can be ambiguous, it is necessary to store a list of nodes (unless one wishes to only pick one possible parse tree); the end result is then a forest of possible parse trees. An alternative formulation employs a second table $B[n,n,r]$ of so-called *backpointers*.

Parsing non-CNF context-free grammars

As pointed out by Lange & Leiß (2009), the drawback of all known transformations into Chomsky normal form is that they can lead to an undesirable bloat in grammar size. Using g to denote the size of the original grammar, the size blow-up in the worst case may range from g^2 to 2^{2g} , depending on the transformation algorithm used. For the use in teaching, they propose a slight generalization of the CYK algorithm, "without compromising efficiency of the algorithm, clarity of its presentation, or simplicity of proofs" (Lange & Leiß 2009).

Parsing weighted context-free grammars

It is also possible to extend the CYK algorithm to parse strings using weighted and stochastic context-free grammars. Weights (probabilities) are then stored in the table P instead of booleans, so $P[i,j,A]$ will contain the minimum weight (maximum probability) that the substring from i to j can be derived from A . Further extensions of the algorithm allow all parses of a string to be enumerated from lowest to highest weight (highest to lowest probability).

Valiant's algorithm

Using Landau symbols, the worst case running time of CYK is $\Theta(n^3 \cdot |G|)$, where n is the length of the parsed string and $|G|$ is the size of the CNF grammar G . This makes it one of the most efficient algorithms for recognizing general context-free languages in practice. Valiant (1975) gave an extension of the CYK algorithm. His algorithm computes the same parsing table as the CYK algorithm; yet he showed that algorithms for efficient multiplication of matrices with 0-1-entries can be utilized for performing this computation.

Using the Coppersmith–Winograd algorithm for multiplying these matrices, this gives an asymptotic worst-case running time of $O(n^{2.38} \cdot |G|)$. However, the constant term hidden by the Big O Notation is so large that the Coppersmith–Winograd algorithm is only worthwhile for matrices that are too large to handle on present-day computers (Knuth 1997). The dependence on efficient matrix multiplication cannot be avoided altogether: Lee (2002) has proved that any parser for context-free grammars working in time $O(n^{3-\varepsilon} \cdot |G|)$ can be effectively converted into an algorithm computing the product of $(n \times n)$ -matrices with 0-1-entries in time $O(n^{3-\varepsilon/3})$.

References

- John Cocke and Jacob T. Schwartz (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.
- T. Kasami (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.
- Daniel H. Younger (1967). Recognition and parsing of context-free languages in time n^3 . *Information and Control* 10(2): 189–208.
- Donald E. Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional; 3rd edition (November 14, 1997). ISBN 978-0201896848. pp. 501.
- Lange, Martin; Leiß, Hans (2009), "To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm" (<http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009>), *Informatica Didactica* (pdf (<http://www.informatica-didactica.de/cmsmadesimple/uploads/Artikel/LangeLeiss2009/LangeLeiss2009.pdf>)) 8
- Sipser, Michael (1997), *Introduction to the Theory of Computation* (1st ed.), IPS, p. 99, ISBN 0-534-94728-X
- Lee, Lillian (2002), "Fast context-free grammar parsing requires fast Boolean matrix multiplication", *Journal of the ACM* 49 (1): 1–15, doi:10.1145/505241.505242.
- Leslie G. Valiant. *General context-free recognition in less than cubic time*. Journal of Computer and System Sciences, 10:2 (1975), 308–314.

External links

- CYK parsing demo in JavaScript (<http://www.diotavelli.net/people/void/demos/cky.html>)
- Interactive Applet from the University of Leipzig to demonstrate the CYK-Algorithm (Site is in german) (<http://www.informatik.uni-leipzig.de/alg/lehre/ss08/AUTO-SPRACHEN/Java-Applets/CYK-Algorithmus.html>)
- Exorciser is a Java application to generate exercises in the CYK algorithm as well as Finite State Machines, Markov algorithms etc (<http://www.swisseduc.ch/compscience/exorciser/>)

Simple precedence grammar

A **simple precedence grammar** is a context-free formal grammar that can be parsed with a simple precedence parser.

Formal definition

$G = (N, \Sigma, P, S)$ is a simple precedence grammar if all the production rules in P comply with the following constraints:

- There are no erasing rules (ϵ -productions)
- There are no useless rules (unreachable symbols or unproductive rules)
- For each pair of symbols $X, Y (X, Y \in (N \cup \Sigma))$ there is only one Wirth-Weber precedence relation.
- G is uniquely invertible

Examples

Example 1

$$S \rightarrow aSSb|c$$

precedence table:

	S	a	b	c	\$	
S	$\dot{=}$	\triangleleft	$\dot{=}$	\triangleleft		
a	$\dot{=}$	\triangleleft		\triangleleft		
b		\triangleright		\triangleright	\triangleright	
c		\triangleright	\triangleright	\triangleright	\triangleright	
\$		\triangleleft		\triangleleft		

Simple precedence parser

In computer science, a **Simple precedence parser** is a type of bottom-up parser for context-free grammars that can be used only by Simple precedence grammars.

The implementation of the parser is quite similar to the generic bottom-up parser. A stack is used to store a viable prefix of a sentential form from a rightmost derivation. Symbols \triangleleft , $\dot{=}$ and \triangleright are used to identify the **pivot**, and to know when to **Shift** or when to **Reduce**.

Implementation

- Compute the Wirth-Weber precedence relationship table.
- Start with a stack with only the **starting marker** \$.
- Start with the string being parsed (**Input**) ended with an **ending marker** \$.
- While not (Stack equals to \$S and Input equals to \$) (S = Initial symbol of the grammar)
 - Search in the table the relationship between Top(stack) and NextToken(Input)
 - if the relationship is $\dot{=}$ or \triangleleft
 - **Shift:**
 - Push(Stack, relationship)
 - Push(Stack, NextToken(Input))
 - RemoveNextToken(Input)
 - if the relationship is \triangleright
 - **Reduce:**
 - SearchProductionToReduce(Stack)
 - RemovePivot(Stack)
 - Search in the table the relationship between the Non terminal from the production and first symbol in the stack (Starting from top)
 - Push(Stack, relationship)
 - Push(Stack, Non terminal)

SearchProductionToReduce (Stack)

- search the **Pivot** in the stack the nearest \triangleleft from the top
- search in the productions of the grammar which one have the same right side than the **Pivot**

Example

Given the language:

```
E --> E + T' | T'  
T' --> T  
T --> T * F | F  
F --> ( E' ) | num  
E' --> E
```

num is a terminal, and the lexer parse any integer as **num**.

and the Parsing table:

	E	E'	T	T'	F	+	*	()	num	\$
E						·			·>		·>
E'									·		
T						·>	·		·>		·>
T'						·>			·>		·>
F						·>	·>		·>		·>
+			·<	··	·<	·<	·<		·<		·<
*						··			·<		·<
(·<	··	··	·<	·<	·<			·<		·<
)						·>	·>		·>		·>
num						·>	·>		·>		·>
\$	·<		·<	·<	·<				·<		·<

STACK	PRECEDENCE	INPUT	ACTION
\$	<	2 * (1 + 3)\$	SHIFT
\$ < 2	>	* (1 + 3)\$	REDUCE (F → num)
\$ < F	>	* (1 + 3)\$	REDUCE (T → F)
\$ < T	=	* (1 + 3)\$	SHIFT
\$ < T = *	<	(1 + 3)\$	SHIFT
\$ < T = * < (<	1 + 3)\$	SHIFT
\$ < T = * < (< 1	>	+ 3)\$	REDUCE 4 times (F → num) (T → F) (T' → T) (E → T')
\$ < T = * < (< E	=	+ 3)\$	SHIFT
\$ < T = * < (< E = +	<	3)\$	SHIFT
\$ < T = * < (< E = + < 3	>)\$	REDUCE 3 times (F → num) (T → F) (T' → T)
\$ < T = * < (< E = + = T	>)\$	REDUCE 2 times (E → E + T) (E' → E)
\$ < T = * < (< E'	=)\$	SHIFT
\$ < T = * < (= E' =)	>	\$	REDUCE (F → (E'))
\$ < T = * = F	>	\$	REDUCE (T → T * F)
\$ < T	>	\$	REDUCE 2 times (T' → T) (E → T')
\$ < E	>	\$	ACCEPT

Operator-precedence grammar

An **operator precedence grammar** is a kind of grammar for formal languages.

Technically, an operator precedence grammar is a context-free grammar that has the property (among others^[1]) that no production has either an empty right-hand side or two adjacent nonterminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar. A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers. Operator-precedence parsers can be constructed for a large class of context-free grammars.

Precedence Relations

Operator precedence grammars rely on the following three precedence relations between the terminals:^[2]

Relation	Meaning
$a <\bullet b$	a yields precedence to b
$a =\bullet b$	a has the same precedence as b
$a \bullet> b$	a takes precedence over b

These operator precedence relations allow to delimit the handles in the right sentential forms: $<\bullet$ marks the left end, $=\bullet$ appears in the interior of the handle, and $\bullet>$ marks the right end. Contrary to other shift-reduce parsers, all nonterminals are considered equal for the purpose of identifying handles.^[3] The relations do not have the same properties as their un-dotted counterparts; e. g. $a =\bullet b$ does not generally imply $b =\bullet a$, and $b \bullet> a$ does not follow from $a <\bullet b$. Furthermore, $a =\bullet a$ does not generally hold, and $a \bullet> a$ is possible.

Let us assume that between the terminals a_i and a_{i+1} there is always exactly one precedence relation. Suppose that $\$$ is the end of the string. Then for all terminals b we define: $\$ <\bullet b$ and $b \bullet> \$$. If we remove all nonterminals and place the correct precedence relation: $<\bullet, =\bullet, \bullet>$ between the remaining terminals, there remain strings that can be analyzed by an easily developed bottom-up parser.

Example

For example, the following operator precedence relations can be introduced for simple expressions:^[4]

	id	+	*	\$
id		$\bullet>$	$\bullet>$	$\bullet>$
+	$<\bullet$	$\bullet>$	$<\bullet$	$\bullet>$
*	$<\bullet$	$\bullet>$	$\bullet>$	$\bullet>$
\$	$<\bullet$	$<\bullet$	$<\bullet$	

They follow from the following facts:^[5]

- $+$ has lower precedence than $*$ (hence $+$ $<\bullet *$ and $* \bullet> +$).
- Both $+$ and $*$ are left-associative (hence $+ \bullet> +$ and $* \bullet> *$).

The input string^[6]

$id_1 + id_2 * id_3$

after adding end markers and inserting precedence relations becomes

$\$ <\bullet id_1 \bullet> + <\bullet id_2 \bullet> * <\bullet id_3 \bullet> \$$

Operator Precedence Parsing

Having precedence relations allows to identify handles as follows:^[7]

- scan the string from left until seeing $\bullet>$
- scan backwards (from right to left) over any $=\bullet$ until seeing $<\bullet$
- everything between the two relations $<\bullet$ and $\bullet>$, including any intervening or surrounding nonterminals, forms the handle

It is generally not necessary to scan the entire sentential form to find the handle.

Operator Precedence Parsing Algorithm^[8]

```

Initialize: Set ip to point to the first symbol of w$  

Repeat:  

    If $ is on the top of the stack and ip points to $ then return  

    else  

        Let a be the top terminal on the stack, and b the symbol pointed to by ip  

        if a < $\bullet$  b or a = $\bullet$  b then  

            push b onto the stack  

            advance ip to the next input symbol  

        else if a  $\bullet$ > b then  

            repeat  

                pop the stack  

            until the top stack terminal is related by < $\bullet$  to the terminal most recently popped  

        else error()  

    end

```

Precedence Functions

An operator precedence parser usually does not store the precedence table with the relations, which can get rather large. Instead, precedence functions f and g are defined.^[9] They map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison: $f(a) < g(b)$ must hold if $a <\bullet b$ holds, etc.

Not every table of precedence relations has precedence functions, but in practice for most grammars such functions can be designed.^[10]

Algorithm for Constructing Precedence Functions^[11]

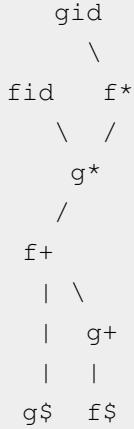
1. Create symbols f_a and g_a for each grammar terminal a and for the end of string symbol;
2. Partition the created symbols in groups so that f_a and g_b are in the same group if $a =\bullet b$ (there can be symbols in the same group even if their terminals are not connected by this relation);
3. Create a directed graph whose nodes are the groups, next for each pair (a,b) of terminals do: place an edge from the group of g_b to the group of f_a if $a <\bullet b$, otherwise if $a \bullet> b$ place an edge from the group of f_a to that of g_b ;
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles, let $f(a)$ be the length of the longest path from the group of f_a and let $g(a)$ be the length of the longest path from the group of g_a .

Example

Consider the following table (repeated from above):^[12]

	id	+	*	\$
id		•>	•>	•>
+	<•	•>	<•	•>
*	<•	•>	•>	•>
\$	<•	<•	<•	

Using the algorithm leads to the following graph:



from which we extract the following precedence functions from the maximum heights in the directed acyclic graph:

	id	+	*	\$
f	4	2	4	0
g	5	1	3	0

Notes

- [1] ASU 1988, p. 203.
- [2] ASU 1988, pp. 203-204.
- [3] ASU 1988, pp. 205-206.
- [4] ASU 1988, p. 205.
- [5] ASU 1988, p. 204.
- [6] ASU 1988, p. 205.
- [7] ASU 1988, p. 205.
- [8] ASU 1988, p. 206.
- [9] ASU 1988, pp. 208-209.
- [10] ASU 1988, p. 209.
- [11] ASU 1988, pp. 209-210.
- [12] ASU 1988, p. 210.

References

- Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. (1988). *Compilers — Principles, Techniques, and Tools*. Addison-Wesley.
- Floyd, R. W. (1963). "Syntactic Analysis and Operator Precedence". *Journal of the ACM* **10**: 316. doi:10.1145/321172.321179.

External links

- Nikolay Nikolaev: IS53011A Language Design and Implementation (<http://homepages.gold.ac.uk/nikolaev/cis324.htm>), Course notes for CIS 324, 2010.

Operator-precedence parser

An **operator precedence parser** is a bottom-up parser that interprets an operator-precedence grammar. For example, most calculators use operator precedence parsers to convert from the human-readable infix notation with order of operations format into an internally optimized computer-readable format like Reverse Polish notation (RPN).

Edsger Dijkstra's shunting yard algorithm is commonly used to implement operator precedence parsers which convert from infix notation to RPN.

Relationship to other parsers

An operator-precedence parser is a simple shift-reduce parser capable of parsing a subset of LR(1) grammars. More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive nonterminals never appear in the right-hand side of any rule.

Operator-precedence parsers are not used often in practice, however they do have some properties that make them useful within a larger design. First, they are simple enough to write by hand, which is not generally the case with more sophisticated shift-reduce parsers. Second, they can be written to consult an operator table at run time, which makes them suitable for languages that can add to or change their operators while parsing.

Perl 6 "sandwiches" an operator-precedence parser in between two Recursive descent parsers in order to achieve a balance of speed and dynamism. This is expressed in the virtual machine for Perl 6, Parrot as the Parser Grammar Engine (PGE). GCC's C and C++ parsers, which are hand-coded recursive descent parsers, are both sped up by an operator-precedence parser that can quickly examine arithmetic expressions.

To show that one grammar is **operator precedence**, first it should be **operator grammar**. Operator precedence grammar is the only grammar which can construct the parse tree even though the given grammar is ambiguous.

Example algorithm known as precedence climbing to parse infix notation

An EBNF grammar that parses infix notation will usually look like this:

```
expression ::= equality-expression
equality-expression ::= additive-expression ( ( '==' | '!=' ) additive-expression ) *
additive-expression ::= multiplicative-expression ( ( '+' | '-' ) multiplicative-expression ) *
multiplicative-expression ::= primary ( ( '*' | '/' ) primary ) *
primary ::= '(' expression ')' | NUMBER | VARIABLE | '-' primary
```

With many levels of precedence, implementing this grammar with a predictive recursive-descent parser can become inefficient. Parsing a number, for example, can require five function calls (one for each non-terminal in the grammar,

until we reach *primary*).

An operator-precedence parser can do the same more efficiently. The idea is that we can left associate the arithmetic operations as long as we find operators with the same precedence, but we have to save a temporary result to evaluate higher precedence operators. The algorithm that is presented here does not need an explicit stack: instead, it uses recursive calls to implement the stack.

The algorithm is not a pure operator-precedence parser like the Dijkstra shunting yard algorithm. It assumes that the *primary* nonterminal is parsed in a separate subroutine, like in a recursive descent parser.

Pseudo-code

The pseudo-code for the algorithm is as follows. The parser starts at function *parse_expression*. Precedence levels are greater or equal to 0.

```

parse_expression ()
    return parse_expression_1 (parse_primary (), 0)

parse_expression_1 (lhs, min_precedence)
    while the next token is a binary operator whose precedence is >= min_precedence
        op := next token
        rhs := parse_primary ()
        while the next token is a binary operator whose precedence is greater
            than op's, or a right-associative operator
            whose precedence is equal to op's
            lookahead := next token
            rhs := parse_expression_1 (rhs, lookahead's precedence)
        lhs := the result of applying op with operands lhs and rhs
    return lhs
```

Example execution of the algorithm

An example execution on the expression $2 + 3 * 4 + 5 == 19$ is as follows. We give precedence 0 to equality expressions, 1 to additive expressions, 2 to multiplicative expressions.

parse_expression_1 (*lhs* = 2, *min_precedence* = 0)

- the next token is +, with precedence 1. the while loop is entered.
- *op* is + (precedence 1)
- *rhs* is 3
- the next token is *, with precedence 2. recursive invocation.

parse_expression_1 (*lhs* = 3, *min_precedence* = 2)

- the next token is *, with precedence 2. the while loop is entered.
 - *op* is * (precedence 2)
 - *rhs* is 4
 - the next token is +, with precedence 1. no recursive invocation.
 - *lhs* is assigned $3*4 = 12$
 - the next token is +, with precedence 1. the while loop is left.
- 12 is returned.
- the next token is +, with precedence 1. no recursive invocation.
- *lhs* is assigned $2+12 = 14$
- the next token is +, with precedence 1. the while loop is not left.

- op is + (precedence 1)
 - rhs is 5
 - the next token is ==, with precedence 0. no recursive invocation.
 - lhs is assigned $14+5 = 19$
 - the next token is ==, with precedence 0. the while loop is not left.
 - op is == (precedence 0)
 - rhs is 19
 - the next token is *end-of-line*, which is not an operator. no recursive invocation.
 - lhs is assigned the result of evaluating $19 == 19$, for example 1 (as in the C standard).
 - the next token is *end-of-line*, which is not an operator. the while loop is left.
- 1 is returned.

Alternatives to Dijkstra's Algorithm

There are alternative ways to apply operator precedence rules which do not involve the Shunting Yard algorithm.

One is to build a tree of the original expression, then apply tree rewrite rules to it.

Such trees do not necessarily need to be implemented using data structures conventionally used for trees. Instead, tokens can be stored in flat structures, such as tables, by simultaneously building a priority list which states what elements to process in which order. As an example, such an approach is used in the *Mathematical Formula Decomposer*.^[1]

Another approach is to first fully parenthesise the expression, inserting a number of parentheses around each operator, such that they lead to the correct precedence even when parsed with a linear, left-to-right parser. This algorithm was used in the early FORTRAN I compiler.

Example code of a simple C application that handles parenthesisation of basic math operators (+, -, *, /, ^ and parentheses):

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    printf("( ( ( (");
    for(i=1;i!=argc;i++) {
        if(argv[i] && !argv[i][1]) {
            switch(*argv[i]) {
                case '(': printf("((("); continue;
                case ')': printf("))))"); continue;
                case '^': printf(")^("); continue;
                case '*': printf(")*("); continue;
                case '/': printf(")/("); continue;
                case '+':
                    if (i == 1 || strchr("(^*/+-", *argv[i-1]))
                        printf("+");
                    else
                        printf(")))+(((");
                    continue;
                case '-':
                    if (i == 1 || strchr("(^*/+-", *argv[i-1]))
                        printf("-");
                    else
                        printf(")))-(((");
                    continue;
            }
        }
    }
    printf("))";
}
```

```
        printf("-");
    else
        printf(") ) - ( ( ");
    continue;
}
}
printf("%s", argv[i]);
}
printf(") ) ) \n");
return 0;
}
```

For example, when compiled and invoked from command line with parameters

```
a * b + c ^ d / e
```

it produces

```
(( ( (a) * ( (b) ) ) + ( ( (c) ^ (d) ) / ( (e) ) ) )
```

as output on the console.

References

[1] Mathematical Formula Decomposer (<http://herbert.gandraxa.com/herbert/mfd.asp>)

External links

- Parsing Expressions by Recursive Descent (http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm)
Theodore Norvell (C) 1999-2001. Access data September 14, 2006.
- Sequential formula translation. (<http://portal.acm.org/citation.cfm?id=366968>) Samelson, K. and Bauer, F. L. 1960. Commun. ACM 3, 2 (Feb. 1960), 76-83. DOI= <http://doi.acm.org/10.1145/366959.366968>

Shunting-yard algorithm

The **shunting-yard algorithm** is a method for parsing mathematical expressions specified in infix notation. It can be used to produce output in Reverse Polish notation (RPN) or as an abstract syntax tree (AST). The algorithm was invented by Edsger Dijkstra and named the "shunting yard" algorithm because its operation resembles that of a railroad shunting yard. Dijkstra first described the Shunting Yard Algorithm in Mathematisch Centrum report MR-35.

Like the evaluation of RPN, the shunting yard algorithm is stack-based. Infix expressions are the form of mathematical notation most people are used to, for instance $3+4$ or $3+4*(2-1)$. For the conversion there are two text variables (strings), the input and the output. There is also a stack that holds operators not yet added to the output queue. To convert, the program reads each symbol in order and does something based on that symbol.

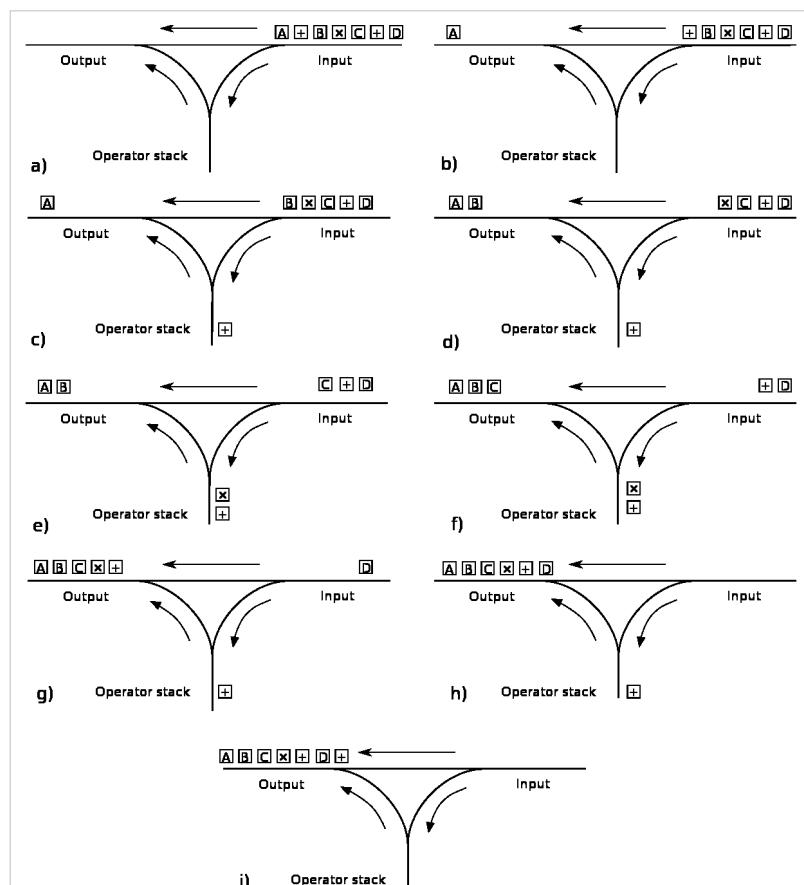
A simple conversion

Input: $3+4$

1. Add 3 to the output queue
(whenever a number is read it is added to the output)
2. Push + (or its ID) onto the operator stack
3. Add 4 to the output queue
4. After reading the expression pop the operators off the stack and add them to the output.
5. In this case there is only one, "+".
6. Output $3\ 4\ +$

This already shows a couple of rules:

- All numbers are added to the output when they are read.
- At the end of reading the expression, pop all operators off the stack and onto the output.



Graphical illustration of algorithm, using a three way railroad junction. The input is processed one symbol at a time, if a variable or number is found it is copied direct to the output b), d), f), h). If the symbol is an operator it is pushed onto the operator stack c), e), however, if its precedence is less than that of the operator at the top of the stack or the precedences are equal and the operator is left associative then that operator is popped off the stack and added to the output g). Finally remaining operators are popped off the stack and added to the output.

The algorithm in detail

- While there are tokens to be read:
 - Read a token.
 - If the token is a number, then add it to the output queue.
 - If the token is a function token, then push it onto the stack.
 - If the token is a function argument separator (e.g., a comma):
 - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue. If no left parentheses are encountered, either the separator was misplaced or parentheses were mismatched.

- If the token is an operator, o_1 , then:
 - while there is an operator token, o_2 , at the top of the stack, and
 - either o_1 is left-associative and its precedence is less than or equal to that of o_2 ,
 - or o_1 is right-associative and its precedence is less than that of o_2 ,
 - pop o_2 off the stack, onto the output queue;
 - push o_1 onto the stack.
- If the token is a left parenthesis, then push it onto the stack.
- If the token is a right parenthesis:
 - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue.
 - Pop the left parenthesis from the stack, but not onto the output queue.
 - If the token at the top of the stack is a function token, pop it onto the output queue.
 - If the stack runs out without finding a left parenthesis, then there are mismatched parentheses.
- When there are no more tokens to read:
 - While there are still operator tokens in the stack:
 - If the operator token on the top of the stack is a parenthesis, then there are mismatched parentheses.
 - Pop the operator onto the output queue.
- Exit.

To analyze the running time complexity of this algorithm, one has only to note that each token will be read once, each number, function, or operator will be printed once, and each function, operator, or parenthesis will be pushed onto the stack and popped off the stack once – therefore, there are at most a constant number of operations executed per token, and the running time is thus $O(n)$ – linear in the size of the input.

The shunting yard algorithm can also be applied to produce prefix notation (also known as polish notation). To do this one would simply start from the beginning of a string of tokens to be parsed and work backwards, and then reversing the output queue (therefore making the output queue an output stack).

Detailed example

Input: $3 + 4 * 2 / (1 - 5) ^ 2 ^ 3$

operator	precedence	associativity
$^$	4	Right
$*$	3	Left
$/$	3	Left
$+$	2	Left
$-$	2	Left

Token	Action	Output (in RPN)	Operator Stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
*	Push token to stack	3 4	* +	* has higher precedence than +
2	Add token to output	3 4 2	* +	
/	Pop stack to output	3 4 2 *	+	/ and * have same precedence
	Push token to stack	3 4 2 *	/ +	/ has higher precedence than +
(Push token to stack	3 4 2 *	(/ +	
1	Add token to output	3 4 2 * 1	(/ +	
-	Push token to stack	3 4 2 * 1	- (/ +	
5	Add token to output	3 4 2 * 1 5	- (/ +	
)	Pop stack to output	3 4 2 * 1 5 -	(/ +	Repeated until "(" found
	Pop stack	3 4 2 * 1 5 -	/ +	Discard matching parenthesis
^	Push token to stack	3 4 2 * 1 5 -	^ / +	^ has higher precedence than /
2	Add token to output	3 4 2 * 1 5 - 2	^ / +	
^	Push token to stack	3 4 2 * 1 5 - 2	^ ^ / +	^ is evaluated right-to-left
3	Add token to output	3 4 2 * 1 5 - 2 3	^ ^ / +	
end	Pop entire stack to output	3 4 2 * 1 5 - 2 3 ^ ^ / +		

If you were writing an interpreter, this output would be tokenized and written to a compiled file to be later interpreted. Conversion from infix to RPN can also allow for easier simplification of expressions. To do this, act like you are solving the RPN expression, however, whenever you come to a variable its value is null, and whenever an operator has a null value, it and its parameters are written to the output (this is a simplification, problems arise when the parameters are operators). When an operator has no null parameters its value can simply be written to the output. This method obviously doesn't include all the simplifications possible: It's more of a constant folding optimization.

C example

```
#include <string.h>
#include <stdio.h>
#define bool int
#define false 0
#define true 1

// operators
// precedence    operators      associativity
// 1             !
// 2             * / %
// 3             + -
// 4             =
int op_preced(const char c)
{
    switch(c) {

```

```
case '!' :
    return 4;
case '*' : case '/' : case '%':
    return 3;
case '+' : case '-':
    return 2;
case '=' :
    return 1;
}
return 0;
}

bool op_left_assoc(const char c)
{
    switch(c) {
        // left to right
        case '*' : case '/' : case '%' : case '+' : case '-':
            return true;
        // right to left
        case '=' : case '!':
            return false;
    }
    return false;
}

unsigned int op_arg_count(const char c)
{
    switch(c) {
        case '*' : case '/' : case '%' : case '+' : case '-' : case '=':
            return 2;
        case '!':
            return 1;
        default:
            return c - 'A';
    }
    return 0;
}

#define is_operator(c) (c == '+' || c == '-' || c == '/' || c == '*'
|| c == '!' || c == '%' || c == '=')
#define is_function(c) (c >= 'A' && c <= 'Z')
#define is_ident(c) ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z'))

bool shunting_yard(const char *input, char *output)
{
    const char *strpos = input, *strend = input + strlen(input);
    char c, *outpos = output;
```

```
char stack[32];           // operator stack
unsigned int sl = 0;      // stack length
char     sc;              // used for record stack element

while(strupos < strand)    {
    // read one token from the input stream
    c = *strupos;
    if(c != ' ')    {
        // If the token is a number (identifier), then add it to
the output queue.
        if(is_ident(c))  {
            *outpos = c; ++outpos;
        }
        // If the token is a function token, then push it onto the
stack.
        else if(is_function(c))   {
            stack[sl] = c;
            ++sl;
        }
        // If the token is a function argument separator (e.g., a
comma):
        else if(c == ',')    {
            bool pe = false;
            while(sl > 0)    {
                sc = stack[sl - 1];
                if(sc == '(')  {
                    pe = true;
                    break;
                }
                else  {
                    // Until the token at the top of the stack is a
left parenthesis,
                    // pop operators off the stack onto the output
queue.
                    *outpos = sc;
                    ++outpos;
                    sl--;
                }
            }
            // If no left parentheses are encountered, either the
separator was misplaced
            // or parentheses were mismatched.
            if(!pe)  {
                printf("Error: separator or parentheses
mismatched\n");
                return false;
            }
        }
    }
}
```

```
        }
    }

    // If the token is an operator, op1, then:
    else if(is_operator(c)) {
        while(sl > 0) {
            sc = stack[sl - 1];
            // While there is an operator token, o2, at the top
            of the stack
            // op1 is left-associative and its precedence is
            less than or equal to that of op2,
            // or op1 is right-associative and its precedence
            is less than that of op2,
            if(is_operator(sc) &&
               ((op_left_assoc(c) && (op_preced(c) <= op_preced(sc))) ||
                (!op_left_assoc(c) && (op_preced(c) < op_preced(sc)))) {
                // Pop o2 off the stack, onto the output queue;
                *outpos = sc;
                ++outpos;
                sl--;
            }
            else {
                break;
            }
        }
        // push op1 onto the stack.
        stack[sl] = c;
        ++sl;
    }

    // If the token is a left parenthesis, then push it onto
    the stack.
    else if(c == '(') {
        stack[sl] = c;
        ++sl;
    }

    // If the token is a right parenthesis:
    else if(c == ')') {
        bool pe = false;
        // Until the token at the top of the stack is a left
        parenthesis,
        // pop operators off the stack onto the output queue
        while(sl > 0) {
            sc = stack[sl - 1];
            if(sc == '(') {
                pe = true;
                break;
            }
            else {
```

```
        *outpos = sc;
        ++outpos;
        sl--;
    }
}

// If the stack runs out without finding a left
parenthesis, then there are mismatched parentheses.
if(!pe) {
    printf("Error: parentheses mismatched\n");
    return false;
}

// Pop the left parenthesis from the stack, but not
onto the output queue.
sl--;
// If the token at the top of the stack is a function
token, pop it onto the output queue.
if(sl > 0) {
    sc = stack[sl - 1];
    if(is_function(sc)) {
        *outpos = sc;
        ++outpos;
        sl--;
    }
}
else {
    printf("Unknown token %c\n", c);
    return false; // Unknown token
}
++strpos;
}

// When there are no more tokens to read:
// While there are still operator tokens in the stack:
while(sl > 0) {
    sc = stack[sl - 1];
    if(sc == '(' || sc == ')') {
        printf("Error: parentheses mismatched\n");
        return false;
    }
    *outpos = sc;
    ++outpos;
    --sl;
}
*outpos = 0; // Null terminator
return true;
}
```

```
bool execution_order(const char *input) {
    printf("order: (arguments in reverse order)\n");
    const char *strpos = input, *strend = input + strlen(input);
    char c, res[4];
    unsigned int sl = 0, sc, stack[32], rn = 0;
    // While there are input tokens left
    while(strpos < strend) {
        // Read the next token from input.
        c = *strpos;
        // If the token is a value or identifier
        if(is_ident(c)) {
            // Push it onto the stack.
            stack[sl] = c;
            ++sl;
        }
        // Otherwise, the token is an operator (operator here
        includes both operators, and functions).
        else if(is_operator(c) || is_function(c)) {
            sprintf(res, "%02d", rn);
            printf("%s = ", res);
            ++rn;
            // It is known a priori that the operator takes n
            arguments.
            unsigned int nargs = op_arg_count(c);
            // If there are fewer than n values on the stack
            if(sl < nargs) {
                // (Error) The user has not input sufficient
                values in the expression.
                return false;
            }
            // Else, Pop the top n values from the stack.
            // Evaluate the operator, with the values as
            arguments.
            if(is_function(c)) {
                printf("%c(", c);
                while(nargs > 0) {
                    sc = stack[sl - 1];
                    sl--;
                    if(nargs > 1) {
                        printf("%s, ", &sc);
                    }
                    else {
                        printf("%s)\n", &sc);
                    }
                    --nargs;
                }
            }
        }
    }
}
```

```
        }

    else      {
        if(nargs == 1) {
            sc = stack[sl - 1];
            sl--;
            printf("%c %s;\n", c, &sc);
        }
        else      {
            sc = stack[sl - 1];
            sl--;
            printf("%s %c ", &sc, c);
            sc = stack[sl - 1];
            sl--;
            printf("%s;\n", &sc);
        }
    }

    // Push the returned results, if any, back onto the
stack.

    stack[sl] = *(unsigned int*)res;
    ++sl;
}

++strpos;
}

// If there is only one value in the stack
// That value is the result of the calculation.

if(sl == 1) {
    sc = stack[sl - 1];
    sl--;
    printf("%s is a result\n", &sc);
    return true;
}

// If there are more values in the stack
// (Error) The user input has too many values.

return false;
}

int main() {
    // functions: A() B(a) C(a, b), D(a, b, c) ...
    // identifiers: 0 1 2 3 ... and a b c d e ...
    // operators: = - + / * %

    const char *input = "a = D(f - b * c + d, !e, g)";
    char output[128];
    printf("input: %s\n", input);
    if(shunting_yard(input, output)) {
        printf("output: %s\n", output);
        if(!execution_order(output))
            printf("\nInvalid input\n");
    }
}
```

```
    }
    return 0;
}
```

This code produces the following output:

```
input: a = D(f - b * c + d, !e, g)
output: afbc*-d+e!gD=
order: (arguments in reverse order)
_00 = c * b;
_01 = _00 - f;
_02 = d + _01;
_03 = ! e;
_04 = D(g, _03, _02)
_05 = _04 = a;
_05 is a result
```

External links

- Dijkstra's Original description of the Shunting yard algorithm ^[1]
- Literate Programs implementation in C ^[2]
- Java Applet demonstrating the Shunting yard algorithm ^[3]
- Silverlight widget demonstrating the Shunting yard algorithm and evaluation of arithmetic expressions ^[4]
- Parsing Expressions by Recursive Descent ^[5] Theodore Norvell © 1999–2001. Access date September 14, 2006.
- Extension to the 'Shunting Yard' algorithm to allow variable numbers of arguments to functions ^[6]
- A Python implementation of the Shunting yard algorithm ^[7]

References

- [1] <http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF>
- [2] [http://en.literateprograms.org/Shunting_yard_algorithm_\(C\)](http://en.literateprograms.org/Shunting_yard_algorithm_(C))
- [3] <http://www.chris-j.co.uk/parsing.php>
- [4] <http://www.codeding.com/?article=11>
- [5] http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
- [6] <http://www.kallisti.net.nz/blog/2008/02/>
extension-to-the-shunting-yard-algorithm-to-allow-variable-numbers-of-arguments-to-functions/
- [7] <http://github.com/ekg/shuntingyard/blob/master/shuntingyard.py>

Chart parser

A **chart parser** is a type of parser suitable for ambiguous grammars (including grammars of natural languages). It uses the dynamic programming approach—partial hypothesized results are stored in a structure called a chart and can be re-used. This eliminates backtracking and prevents a combinatorial explosion.

Chart parsing was developed by Martin Kay.

Types of chart parsers

A common approach is to use a variant of the Viterbi algorithm. The Earley parser is a type of chart parser mainly used for parsing in computational linguistics, named for its inventor. Another chart parsing algorithm is the Cocke-Younger-Kasami (CYK) algorithm.

Chart parsers can also be used for parsing computer languages. Earley parsers in particular have been used in compiler compilers where their ability to parse using arbitrary Context-free grammars eases the task of writing the grammar for a particular language. However their lower efficiency has led to people avoiding them for most compiler work.

In bidirectional chart parsing, edges of the chart are marked with a direction, either forwards or backwards, and rules are enforced on the direction in which edges must point in order to be combined into further edges.

In incremental chart parsing, the chart is constructed incrementally as the text is edited by the user, with each change to the text resulting in the minimal possible corresponding change to the chart.

We can distinguish top-down and bottom-up chart parsers, and active and passive chart parsers.

Parsing ambiguity in natural languages

The most serious problem faced by parsers is the ambiguity of natural languages.

Earley parser

The **Earley parser** is a type of chart parser mainly used for parsing in computational linguistics, named after its inventor, Jay Earley. The algorithm uses dynamic programming.

Earley parsers are appealing because they can parse all context-free languages. The Earley parser executes in cubic time ($O(n^3)$, where n is the length of the parsed string) in the general case, quadratic time ($O(n^2)$) for unambiguous grammars, and linear time for almost all LR(k) grammars. It performs particularly well when the rules are written left-recursively.

The algorithm

In the following descriptions, α , β , and γ represent any string of terminals/nonterminals (including the empty string), X and Y represent single nonterminals, and a represents a terminal symbol.

Earley's algorithm is a top-down dynamic programming algorithm. In the following, we use Earley's dot notation: given a production $X \rightarrow \alpha\beta$, the notation $X \rightarrow \alpha \cdot \beta$ represents a condition in which α has already been parsed and β is expected.

For every input position (which represents a position *between* tokens), the parser generates an ordered *state set*. Each state is a tuple $(X \rightarrow \alpha \cdot \beta, i)$, consisting of

- the production currently being matched ($X \rightarrow \alpha \beta$)
- our current position in that production (represented by the dot)
- the position i in the input at which the matching of this production began: the *origin position*

(Earley's original algorithm included a look-ahead in the state; later research showed this to have little practical effect on the parsing efficiency, and it has subsequently been dropped from most implementations.)

The state set at input position k is called $S(k)$. The parser is seeded with $S(0)$ consisting of only the top-level rule. The parser then iteratively operates in three stages: *prediction*, *scanning*, and *completion*.

- **Prediction:** For every state in $S(k)$ of the form $(X \rightarrow \alpha \cdot Y \beta, j)$ (where j is the origin position as above), add $(Y \rightarrow \cdot \gamma, k)$ to $S(k)$ for every production in the grammar with Y on the left-hand side ($Y \rightarrow \gamma$).
- **Scanning:** If a is the next symbol in the input stream, for every state in $S(k)$ of the form $(X \rightarrow \alpha \cdot a \beta, j)$, add $(X \rightarrow \alpha a \cdot \beta, j)$ to $S(k+1)$.
- **Completion:** For every state in $S(k)$ of the form $(X \rightarrow \gamma \cdot, j)$, find states in $S(j)$ of the form $(Y \rightarrow \alpha \cdot X \beta, i)$ and add $(Y \rightarrow \alpha X \cdot \beta, i)$ to $S(k)$.

These steps are repeated until no more states can be added to the set. The set is generally implemented as a queue of states to process (though a given state must appear in one place only), and performing the corresponding operation depending on what kind of state it is.

Pseudocode

Adapted from Speech and Language Processing [1] by Daniel Jurafsky and James H. Martin:

```
function EARLEY-PARSE(words, grammar)
    ENQUEUE(( $Y \rightarrow \cdot S$ , [0,0]), chart[0])
    for i ← from 0 to LENGTH(words) do
        for each state in chart[i] do
            if INCOMPLETE?(state) then
                if NEXT-CAT(state) is a nonterminal then
                    PREDICTOR(state)           // non-terminal
```

```

        else do
            SCANNER(state)           // terminal
        else do
            COMPLETER(state)
        end
    end
    return chart

procedure PREDICTOR((A → a•B, [i, j])),
    for each (B → y) in GRAMMAR-RULES-FOR(B, grammar) do
        ENQUEUE((B → •y, [j, j]), chart[j])
    end

procedure SCANNER((A → a•B, [i, j])),
    if B ∈ PARTS-OF-SPEECH(word[j]) then
        ENQUEUE((B → word[j], [j, j + 1]), chart[j + 1])
    end

procedure COMPLETER((B → y•, [j, k])),
    for each (A → a•Bβ, [i, j]) in chart([j]) do
        ENQUEUE((A → aB•β, [i, k]), chart[k])
    end

```

Example

Consider the following simple grammar for arithmetic expressions:

```

P → S      # the start rule
S → S + M
| M
M → M * T
| T
T → number

```

With the input:

```
2 + 3 * 4
```

This is the sequence of state sets:

```

(state no.) Production (Origin) # Comment
-----
== S(0): •2 + 3 * 4 ==
(1) P → •S          (0) # start rule
(2) S → •S + M     (0) # predict from (1)
(3) S → •M          (0) # predict from (1)
(4) M → •M * T     (0) # predict from (3)
(5) M → •T          (0) # predict from (3)
(6) T → •number     (0) # predict from (5)

```

```

== S(1): 2 • + 3 * 4 ==
(1) T → number • (0)      # scan from S(0)(6)
(2) M → T • (0)      # complete from S(0)(5)
(3) M → M • * T (0)      # complete from S(0)(4)
(4) S → M • (0)      # complete from S(0)(3)
(5) S → S • + M (0)      # complete from S(0)(2)
(6) P → S • (0)      # complete from S(0)(1)

== S(2): 2 + • 3 * 4 ==
(1) S → S + • M (0)      # scan from S(1)(5)
(2) M → • M * T (2)      # predict from (1)
(3) M → • T (2)      # predict from (1)
(4) T → • number (2)      # predict from (3)

== S(3): 2 + 3 • * 4 ==
(1) T → number • (2)      # scan from S(2)(4)
(2) M → T • (2)      # complete from S(2)(3)
(3) M → M • * T (2)      # complete from S(2)(2)
(4) S → S + M • (0)      # complete from S(2)(1)
(5) S → S • + M (0)      # complete from S(0)(2)
(6) P → S • (0)      # complete from S(0)(1)

== S(4): 2 + 3 * • 4 ==
(1) M → M * • T (2)      # scan from S(3)(3)
(2) T → • number (4)      # predict from (1)

== S(5): 2 + 3 * 4 • ==
(1) T → number • (4)      # scan from S(4)(2)
(2) M → M * T • (2)      # complete from S(4)(1)
(3) M → M • * T (2)      # complete from S(2)(2)
(4) S → S + M • (0)      # complete from S(2)(1)
(5) S → S • + M (0)      # complete from S(0)(2)
(6) P → S • (0)      # complete from S(0)(1)

```

The state ($P \rightarrow S \bullet, 0$) represents a completed parse. This state also appears in $S(3)$ and $S(1)$, which are complete sentences.

References

- J. Earley, "An efficient context-free parsing algorithm" ^[2], *Communications of the Association for Computing Machinery*, **13**:2:94-102, 1970.
- J. Leo, A general context-free parsing algorithm running in linear time on every $LR(k)$ grammar without using look-ahead, *Theoretical Computer Science*, **82**:165-176, 1991.
- J. Aycock and R.N. Horspool. Practical Earley Parsing ^[3]. *The Computer Journal*, **45**:6:620-630, 2002.
- Daniel M. Roberts Earley Parsing for Context-Sensitive Grammars ^[4]

External links

C Implementations

- 'early' [5] An Earley parser C -library.
- 'C Earley Parser' [6] An Earley parser C.

Java Implementations

- PEN [7] A Java library that implements the Earley.
- Pep [8] A Java library that implements the Earley algorithm and provides charts and parse trees as parsing artifacts.
- [9] A Java implementation of Earley parser.

Perl Implementations

- Marpa [10] A Perl module, incorporating improvements made to the Earley algorithm by Joop Leo, and by Aycock and Horspool.
- Parse::Earley [11] An Perl module that implements Jay Earley's original algorithm.

Python Implementations

- Charty [12] a Python implementation of an Earley parser.
- NLTK [13] a Python toolkit that has an Earley parser.
- Spark [14] an Object Oriented "little language framework" for Python that implements an Earley parser.

Common Lisp Implementations

- CL-EARLEY-PARSER [15] A Common Lisp library that implements an Earley parser.

Scheme/Racket Implementations

- Charty-Racket [16] A Scheme / Racket implementation of an Earley parser.

Resources

- The Accent compiler-compiler [17]

References

- [1] <http://www.cs.colorado.edu/~martin/slp2.html>
- [2] <http://portal.acm.org/citation.cfm?doid=362007.362035>
- [3] <http://www.cs.uvic.ca/~nigelh/Publications/PracticalEarleyParsing.pdf>
- [4] <http://danielmattosroberts.com/earley/context-sensitive-earley.pdf>
- [5] <http://cocom.sourceforge.net/ammunition-13.html>
- [6] <https://bitbucket.org/abki/c-earley-parser/src>
- [7] <http://linguateca.dei.uc.pt/index.php?sep=recursos>
- [8] <http://www.ling.ohio-state.edu/~scott/#projects-pep>
- [9] <http://www.cs.umanitoba.ca/~comp4190/Earley/Earley.java>
- [10] <http://search.cpan.org/dist/Marpa/>
- [11] <http://search.cpan.org/~lpalmer/Parse-Earley-0.15/Earley.pm>
- [12] <http://www.cavar.me/damir/charty/python/>
- [13] <http://nltk.sourceforge.net/>
- [14] <http://pages.cpsc.ucalgary.ca/~aycock/spark/>
- [15] <http://www.cliki.net/CL-EARLEY-PARSER>
- [16] <http://www.cavar.me/damir/charty/scheme/>

[17] <http://accent.compilertools.net/Entire.html>

The lexer hack

When parsing computer programming languages, **the lexer hack** (as opposed to "a lexer hack") describes a common solution to the problems which arise when attempting to use a regular grammar-based lexer to classify tokens in ANSI C as either variable names or type names.

Problem

In a compiler, the lexer performs one of the earliest stages of converting the source code to a program. It scans the text to extract meaningful *tokens*, such as words, numbers, and strings. The parser analyzes sequences of tokens attempting to match them to syntax rules representing language structures, such as loops and variable declarations. A problem occurs here if a single sequence of tokens can ambiguously match more than one syntax rule.

This ambiguity can happen in C if the lexer does not distinguish between variable and typedef identifiers.^[1] For example, in the C expression:

(A) * B

the lexer may find these tokens:

1. left parenthesis
2. identifier 'A'
3. right parenthesis
4. operator '*'^[2]
5. identifier 'B'

The parser can interpret this as variable *A* multiplied by *B* or as type *A* casting the dereferenced value of *B*. This is known as the "typedef-name: identifier" problem.

The hack solution

The solution generally consists of feeding information from the parser's symbol table back into the lexer. This incestuous mixing of the lexer and parser is generally regarded as inelegant, which is why it is called a "hack". The lexer cannot distinguish type identifiers from other identifiers without extra context because all identifiers have the same format.

With the hack in the above example, when the lexer finds the identifier *A* it should be able to classify the token as a type identifier. The rules of the language would be clarified by specifying that typecasts require a type identifier and the ambiguity disappears.

The problem also exists in C++ and parsers can use the same hack.^[1]

Alternative solutions

This problem does not arise (and hence needs no "hack" in order to solve) when using lexerless parsing techniques.

Some parser generators, such as the yacc-derived BtYacc ("Backtracking Yacc"), give the generated parser the ability to try multiple attempts to parse the tokens. In the problem described here, if an attempt fails because of semantic information about the identifier, it can backtrack and attempt other rules.^[3]

References

- [1] Roskind, James A. (1991-07-11). "A YACC-able C++ 2.1 GRAMMAR, AND THE RESULTING AMBIGUITIES" (<http://www.cs.utah.edu/research/projects/ms0/goofie/grammar5.txt>). .
- [2] Bendersky, Eli (2007-11-24). "The context sensitivity of C's grammar" (<http://eli.thegreenplace.net/2007/11/24/the-context-sensitivity-of-cs-grammar/>). .
- [3] "BtYacc 3.0" (<http://www.siber.com/btyacc/>). . Based on yacc with modifications by Chris Dodd and Vadim Maslov.

Citations

- <http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elkhound/index.html>
- <http://cs.nyu.edu/rgrimm/papers/pldi06.pdf>
- <http://cens.ioc.ee/local/man/CompaqCompilers/ladebug/ladebug-manual-details.html>
- <http://www.springerlink.com/index/YN4GQ2YMNQUY693L.pdf>
- <http://news.gmane.org/find-root.php?group=gmane.comp.lang.groovy.jsr&article=843&type=blog>
- http://groups.google.com/group/comp.compilers/browse_frm/thread/db7f68e9d8b49002/fa20bf5de9c73472?lnk=st&q=%2B%22the+lexer+hack%22&rnum=1&hl=en#fa20bf5de9c73472

Scannerless parsing

Scannerless parsing (also called **lexerless parsing**) refers to the use of a single formalism to express both the lexical and context-free syntax used to parse a language.

This parsing strategy is suitable when a clear lexer–parser distinction is unneeded. Examples of when this is appropriate include TeX, most wiki grammars, makefiles, and simple per application control languages.

Advantages

- Only **one metasyntax** is needed
- **Non-regular lexical structure** is handled easily
- "**Token classification**" is **unneeded** which removes the need for design accommodations such as "the lexer hack" and language keywords (such as "while" in C)
- Grammars can be **compositional** (can be merged without human intervention)^{compositional}

Disadvantages

- Since the lexical scanning and syntactic parsing processing is combined, the resulting parser tends to be **harder to understand and debug** for more complex languages
- Most parsers of character-level grammars are **nondeterministic**
- There is **no guarantee** that the language being parsed is **unambiguous**

Required extensions

Unfortunately, when parsed at the character level, most popular programming languages are no longer strictly context-free. Visser identified five key extensions to classical context-free syntax which handle almost all common non-context-free constructs arising in practice:

- **Follow restrictions**, a limited form of "longest match"
- **Reject productions**, a limited form of negative matching (as found in boolean grammars)
- **Preference attributes** to handle the dangling else construct in C-like languages
- *Per-production transitions* rather than per-nonterminal transitions in order to facilitate:
 - **Associativity attributes**, which prevent a self-reference in a particular production of a nonterminal from producing that same production
 - **Precedence/priority rules**, which prevent self-references in higher-precedence productions from producing lower-precedence productions

Implementations

- SGLR ^[1] is a parser for the modular Syntax Definition Formalism SDF, and is part of the ASF+SDF Meta-Environment and the Stratego/XT program transformation system.
- JSGLR ^[2], a pure Java implementation of SGLR, also based on SDF.
- TXL supports character-level parsing.
- dparser ^[3] generates ANSI C code for scannerless GLR parsers.
- Spirit allows for both scannerless and scanner-based parsing.
- SBP is a scannerless parser for boolean grammars (a superset of context-free grammars), written in Java.
- Laja ^[4] is a two phase scannerless parser generator with support for mapping the grammar rules into objects, written in Java.
- Wormhole ^[5] has an option to generate scannerless GLR in various languages.

Notes

- This is because parsing at the character level makes the language recognized by the parser a single context-free language defined on characters, as opposed to a context-free language of sequences of strings in regular languages. Some lexerless parsers handle the entire class of context-free languages, which is closed under composition.

Further reading

Visser, E. (1997b). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam ^[6]

References

- [1] <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/SGLR>
- [2] <http://strategoxt.org/Stratego/JSGLR>
- [3] <http://dparser.sourceforge.net/>
- [4] <http://laja.sourceforge.net/>
- [5] <http://www.mightyheave.com/>
- [6] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.7828>

Semantic analysis

Attribute grammar

An **attribute grammar** is a formal way to define attributes for the productions of a formal grammar, associating these attributes to values. The evaluation occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or compiler.

The attributes are divided into two groups: *synthesized* attributes and *inherited* attributes. The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes. The inherited attributes are passed down from parent nodes.

In some approaches, synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it. For instance, when constructing a language translation tool, such as a compiler, it may be used to assign semantic values to syntax constructions. Also, it is possible to validate semantic checks associated with a grammar, representing the rules of a language not explicitly imparted by the syntax definition.

Attribute grammars can also be used to translate the syntax tree directly into code for some specific machine, or into some intermediate language.

One strength of attribute grammars is that they can transport information from anywhere in the abstract syntax tree to anywhere else, in a controlled and formal way.

Example

The following is a simple Context-free grammar which can describe a language made up of multiplication and addition of integers.

```

Expr → Expr + Term
Expr → Term
Term → Term * Factor
Term → Factor
Factor → "(" Expr ")"
Factor → integer

```

The following attribute grammar can be used to calculate the result of an expression written in the grammar. Note that this grammar only uses synthesized values, and is therefore an S-attributed grammar.

```

Expr1 → Expr2 + Term [ Expr1.value = Expr2.value + Term.value ]
Expr → Term [ Expr.value = Term.value ]
Term1 → Term2 * Factor [ Term1.value = Term2.value * Factor.value ]
Term → Factor [ Term.value = Factor.value ]
Factor → "(" Expr ")" [ Factor.value = Expr.value ]
Factor → integer [ Factor.value = strToInt(integer.str) ]

```

Synthesized attributes

Let $G = \langle V_n, V_t, P, S \rangle$ be an Attribute grammar, where

- V_n is the set of non terminal symbols
- V_t is the set of terminal symbols
- P is the set of productions
- S is the distinguished symbol, that is the start symbol

$A.a$ is a synthesized attribute if,

- $A \rightarrow \alpha \in P$
- $\alpha = \alpha_1 \dots \alpha_n, \forall_{i, 1 \leq i \leq n} : \alpha_i \in (V_n \cup V_t)$
- $\{\alpha_{j1}, \dots, \alpha_{jm}\} \subseteq \{\alpha_1, \dots, \alpha_n\}$
- $A.a = f(\alpha_{j1}.a_1, \dots, \alpha_{jm}.a_m)$

Special types of attribute grammars

- S-attributed grammar : a simple type of attribute grammar, using only *synthesized attributes*, but no *inherited attributes*.
- L-attributed grammar : *inherited attributes* can be evaluated in top-down parsing.
- LR-attributed grammar : an L-attributed grammar whose *inherited attributes* can also be evaluated in bottom-up parsing.
- ECLR-attributed grammar : equivalent classes can also be used to optimize the evaluation of inherited attributes.

External links

- Why Attribute Grammars Matter ^[1], The Monad Reader, Issue 4, July 5, 2005. (This article narrates on how the formalism of attribute grammars brings aspect-oriented programming to functional programming by helping writing catamorphisms compositionally. It refers to the Utrecht University Attribute Grammar ^[2] system as the implementation used in the examples.)
- Attribute grammar ^[3] in relation to Haskell and functional programming.
- *Semantics of context-free languages*, by Don Knuth, is the original paper introducing attributed grammars.
- D. E. Knuth: The genesis of attribute grammars ^[4]. *Proceedings of the international conference on Attribute grammars and their applications* (1990), LNCS, vol. 461 ^[5], 1–12. Some informal, historical information.
- Jukka Paakkis: Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys* 27:2 (June 1995), 196–255.

References

- [1] http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue4/Why_Attribute_Grammars_Matter
- [2] <http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem>
- [3] http://www.haskell.org/haskellwiki/Attribute_grammar
- [4] <http://www-cs-faculty.stanford.edu/~knuth/papers/gag.tex.gz>
- [5] <http://www.springerlink.com/content/w35h4669q16j/>

L-attributed grammar

L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing. Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar. These are comparable with S-attributed grammars. Used for code synthesis.

LR-attributed grammar

LR-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated on LR parsing. As a result, attribute evaluation in LR-attributed grammars can be incorporated conveniently in bottom-up parsing. zyacc is based on LR-attributed grammars. They are a subset of the L-attributed grammars, where the attributes can be evaluated in one left-to-right traversal of the abstract syntax tree. They are a superset of the S-attributed grammars, which allow only synthesized attributes. In yacc, a common hack is to use global variables to simulate some kind of inherited attributes and thus LR-attribution.

External links

- http://www.cs.binghamton.edu/~zdu/zyacc/doc/zyacc_4.html
- Reinhard Wilhelm: LL- and LR-Attributed Grammars. *Programmiersprachen und Programmentwicklung*, 7. *Fachtagung, veranstaltet vom Fachausschuß 2 der GI* (1982), 151–164, Informatik-Fachberichte volume 53.
- J. van Katwijk: A preprocessor for YACC or A poor man's approach to parsing attributed grammars. *Sigplan Notices* **18**:10 (1983), 12–15.

S-attributed grammar

S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes, but only synthesized attributes. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, are a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created *after* creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing.

Specifications for parser generators in the Yacc family can be broadly considered S-attributed grammars. However, these parser generators usually include the capacity to reference global variables and/or fields from within any given grammar rule, meaning that this is not a *pure* S-attributed approach.

Any S-attributed grammar is also an L-attributed grammar.

ECLR-attributed grammar

ECLR-attributed grammars are a special type of attribute grammars. They are a variant of LR-attributed grammars where an equivalence relation on inherited attributes is used to optimize attribute evaluation. EC stands for equivalence class. Rie is based on ECLR-attributed grammars. They are a superset of LR-attributed grammars.

External links

- <http://www.is.titech.ac.jp/~sassa/lab/rie-e.html>
- M. Sassa, H. Ishizuka and I. Nakata: ECLR-attributed grammars: a practical class of LR-attributed grammars. *Inf. Process. Lett.* **24** (1987), 31–41.
- M. Sassa, H. Ishizuka and I. Nakata: Rie, a Compiler Generator Based on a One-pass-type Attribute Grammar^[1]. *Software—practice and experience* **25**:3 (March 1995), 229–250.

References

[1] <http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol25/issue3/spe947.pdf>

Intermediate representation

In computer science, an **intermediate language** is the language of an abstract machine designed to aid in the analysis of computer programs. The term comes from their use in compilers, where a compiler first translates the source code of a program into a form more suitable for code-improving transformations, as an intermediate step before generating object or machine code for a target machine. The design of an intermediate language typically differs from that of a practical machine language in three fundamental ways:

- Each instruction represents exactly one fundamental operation; *e.g.* "shift-add" addressing modes common in microprocessors are not present.
- Control flow information may not be included in the instruction set.
- The number of registers available may be large, even limitless.

A popular format for intermediate languages is three address code.

A variation in the meaning of this term is to refer to those languages used as an intermediate language by some high-level programming languages which do not output object or machine code, but output the intermediate language only, to submit to a compiler for such language, which then outputs finished object or machine code. This is usually done to gain optimization much as treated above, or portability by using an intermediate language that has compilers for many processors and operating systems, such as C. Languages used for this fall in complexity between high-level languages and low-level languages, such as assembly languages.

Intermediate representation

An **intermediate representation** (IR) is a data structure that is constructed from input data to a program, and from which part or all of the output data of the program is constructed in turn. Use of the term usually implies that most of the information present in the input is retained by the intermediate representation, with further annotations or rapid lookup features.

A canonical example is found in most modern compilers, where the linear human-readable text representing a program is transformed into an intermediate graph data structure that allows flow analysis and re-arrangements before starting to create the list of actual CPU instructions that will do the work. Use of an intermediate representation allows compiler systems like LLVM to be targeted by many different source languages, and support generation for many different target architectures.

Languages

Though not explicitly designed as an intermediate language, C's nature as an abstraction of assembly and its ubiquity as the de-facto system language in Unix-like and other operating systems has made it a popular intermediate language: Eiffel, Sather, Esterel, some dialects of Lisp (Lush, Gambit), Haskell (Glasgow Haskell Compiler), Squeak's C-subset Slang, Cython, Vala, and others make use of C as an intermediate language. Variants of C have been designed to provide C's features as a portable assembly language, including one of the two languages called C--, the C Intermediate Language and the Low Level Virtual Machine.

Sun Microsystem's Java bytecode is the intermediate language used by all compilers targeting the Java Virtual Machine. The JVM can then do just-in-time compilation to get executable machine code to improve performances. Similarly, Microsoft's Common Intermediate Language is an intermediate language designed to be shared by all compilers for the .NET Framework, before static or dynamic compilation to machine code.

The GNU Compiler Collection (GCC) uses internally several intermediate languages to simplify portability and cross-compilation. Among these languages are

- the historical Register Transfer Language (RTL)

- the tree language GENERIC
- the SSA-based GIMPLE.

While most intermediate languages are designed to support statically typed languages, the Parrot intermediate representation is designed to support dynamically typed languages—initially Perl and Python.

The ILOC intermediate language^[1] is used in classes on compiler design as a simple target language.^[2]

References

- [1] "An ILOC Simulator" (<http://www.engr.sjsu.edu/wbarrett/Parser/simManual.htm>) by W. A. Barrett 2007, paraphrasing Keith Cooper and Linda Torczon, "Engineering a Compiler", Morgan Kaufmann, 2004. ISBN 1-55860-698-X.
[2] "CISC 471 Compiler Design" (<http://www.cis.udel.edu/~pollock/471/project2spec.pdf>) by Uli Kremer

External links

- The Stanford SUIF Group (<http://suif.stanford.edu/>)

Intermediate language

In computer science, an **intermediate language** is the language of an abstract machine designed to aid in the analysis of computer programs. The term comes from their use in compilers, where a compiler first translates the source code of a program into a form more suitable for code-improving transformations, as an intermediate step before generating object or machine code for a target machine. The design of an intermediate language typically differs from that of a practical machine language in three fundamental ways:

- Each instruction represents exactly one fundamental operation; *e.g.* "shift-add" addressing modes common in microprocessors are not present.
- Control flow information may not be included in the instruction set.
- The number of registers available may be large, even limitless.

A popular format for intermediate languages is three address code.

A variation in the meaning of this term is to refer to those languages used as an intermediate language by some high-level programming languages which do not output object or machine code, but output the intermediate language only, to submit to a compiler for such language, which then outputs finished object or machine code. This is usually done to gain optimization much as treated above, or portability by using an intermediate language that has compilers for many processors and operating systems, such as C. Languages used for this fall in complexity between high-level languages and low-level languages, such as assembly languages.

Intermediate representation

An **intermediate representation** (IR) is a data structure that is constructed from input data to a program, and from which part or all of the output data of the program is constructed in turn. Use of the term usually implies that most of the information present in the input is retained by the intermediate representation, with further annotations or rapid lookup features.

A canonical example is found in most modern compilers, where the linear human-readable text representing a program is transformed into an intermediate graph data structure that allows flow analysis and re-arrangements before starting to create the list of actual CPU instructions that will do the work. Use of an intermediate representation allows compiler systems like LLVM to be targeted by many different source languages, and support generation for many different target architectures.

Languages

Though not explicitly designed as an intermediate language, C's nature as an abstraction of assembly and its ubiquity as the de-facto system language in Unix-like and other operating systems has made it a popular intermediate language: Eiffel, Sather, Esterel, some dialects of Lisp (Lush, Gambit), Haskell (Glasgow Haskell Compiler), Squeak's C-subset Slang, Cython, Vala, and others make use of C as an intermediate language. Variants of C have been designed to provide C's features as a portable assembly language, including one of the two languages called C--, the C Intermediate Language and the Low Level Virtual Machine.

Sun Microsystem's Java bytecode is the intermediate language used by all compilers targeting the Java Virtual Machine. The JVM can then do just-in-time compilation to get executable machine code to improve performances. Similarly, Microsoft's Common Intermediate Language is an intermediate language designed to be shared by all compilers for the .NET Framework, before static or dynamic compilation to machine code.

The GNU Compiler Collection (GCC) uses internally several intermediate languages to simplify portability and cross-compilation. Among these languages are

- the historical Register Transfer Language (RTL)
- the tree language GENERIC
- the SSA-based GIMPLE.

While most intermediate languages are designed to support statically typed languages, the Parrot intermediate representation is designed to support dynamically typed languages—initially Perl and Python.

The ILOC intermediate language^[1] is used in classes on compiler design as a simple target language.^[2]

References

- [1] "An ILOC Simulator" (<http://www.engr.sjsu.edu/wbarrett/Parser/simManual.htm>) by W. A. Barrett 2007, paraphrasing Keith Cooper and Linda Torczon, "Engineering a Compiler", Morgan Kaufmann, 2004. ISBN 1-55860-698-X.
[2] "CISC 471 Compiler Design" (<http://www.cis.udel.edu/~pollock/471/project2spec.pdf>) by Uli Kremer

External links

- The Stanford SUIF Group (<http://suif.stanford.edu/>)

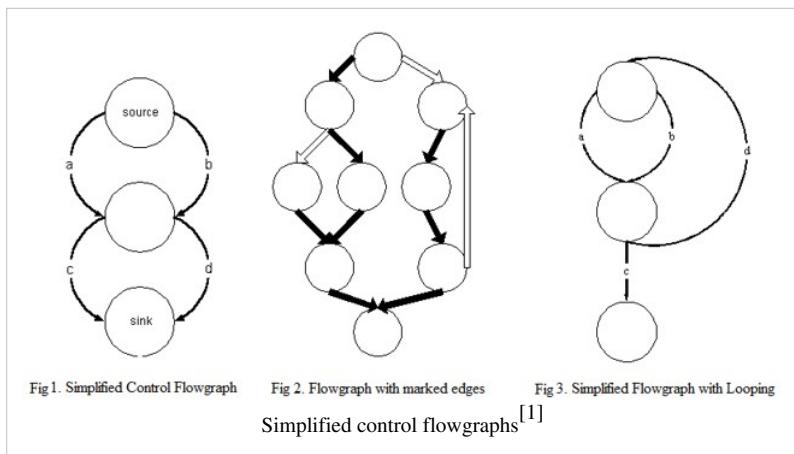
Control flow graph

A **control flow graph (CFG)** in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

Overview

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all control flow leaves.

The CFG is essential to many compiler optimizations and static analysis tools.



Example

Consider the following fragment of code:

```

0: (A) t0 = read_num
1: (A) if t0 mod 2 == 0
2: (B)   print t0 + " is odd."
3: (B)   goto 5
4: (C)   print t0 + " is even."
5: (D) end program
    
```

In the above, we have 4 basic blocks: A from 0 to 1, B from 2 to 3, C at 4 and D at 5. In particular, in this case, A is the "entry block", D the "exit block" and lines 4 and 5 are jump targets. A graph for this fragment has edges from A to B, A to C, B to D and C to D.

Reachability

Reachability is another graph property useful in optimization.

If a block/subgraph is not connected from the subgraph containing the entry block, that block is unreachable during any execution, and so is unreachable code; it can be safely removed.

If the exit block is unreachable from the entry block, it indicates an infinite loop. Not all infinite loops are detectable, of course; see Halting problem.

Dead code and some infinite loops are possible even if the programmer didn't explicitly code that way: optimizations like constant propagation and constant folding followed by jump threading could collapse multiple basic blocks into one, cause edges to be removed from a CFG, etc., thus possibly disconnecting parts of the graph.

Domination relationship

A block M *dominates* a block N if every path from the entry that reaches block N has to pass through block M. The entry block dominates all blocks.

In the reverse direction, block M *postdominates* block N if every path from N to the exit has to pass through block M. The exit block postdominates all blocks.

It is said that a block M *immediately dominates* block N if M dominates N, and there is no intervening block P such that M dominates P and P dominates N. In other words, M is the last dominator on all paths from entry to N. Each block has a unique immediate dominator.

Similarly, there is a notion of *immediate postdominator*: Analogous to *immediate dominator*.

The *dominator tree* is an ancillary data structure depicting the dominator relationships. There is an arc from Block M to Block N if M is an immediate dominator of N. This graph is a tree, since each block has a unique immediate dominator. This tree is rooted at the entry block. Can be calculated efficiently using Lengauer-Tarjan's algorithm.

A *postdominator tree* is analogous to the *dominator tree*. This tree is rooted at the exit block.

Special edges

For practical reasons, it is necessary to introduce some artificial kinds of edges or to process differently some kinds of edges.

A *back edge* is an edge that points to a block that has already been met during a depth-first (DFS) traversal of the graph. Back edges are typical of loops.

A *critical edge* is an edge which is neither the only edge leaving its source block, nor the only edge entering its destination block. These edges must be *split*: a new block must be created in the middle of the edge, in order to insert computations on the edge without affecting any other edges.

An *abnormal edge* is an edge whose destination is unknown. Exception handling constructs can produce them. These edges tend to inhibit optimization.

An *impossible edge* (also known as a *fake edge*) is an edge which has been added to the graph solely to preserve the property that the exit block postdominates all blocks. It cannot ever be traversed.

Loop management

A *loop header* (sometimes called the *entry point* of the loop) is a dominator that is the target of a loop-forming back edge. The loop header dominates all blocks in the loop body.

Suppose block M is a dominator with several incoming edges, some of them being back edges (so M is a loop header). It is advantageous to several optimization passes to break M up into two blocks M_{pre} and M_{loop} . The contents of M and back edges are moved to M_{loop} , the rest of the edges are moved to point into M_{pre} , and a new edge from M_{pre} to M_{loop} is inserted (so that M_{pre} is the immediate dominator of M_{loop}). In the beginning, M_{pre} would be empty, but passes like loop-invariant code motion could populate it. M_{pre} is called the *loop pre-header*, and M_{loop} would be the loop header.

Notes

- [1] Joseph Poole, NIST (1991). A Method to Determine a Basis Set of Paths to Perform Program Testing (<http://hissa.nist.gov/publications/nistir5737/index.html>).

External links

- The Machine-SUIF Control Flow Graph Library (<http://www.eecs.harvard.edu/hube/software/nci/cfg.html>)
- GNU Compiler Collection Internals (<http://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html>)
- Paper " Infrastructure for Profile Driven Optimizations in GCC Compiler (<http://www.ucw.cz/~hubicka/papers/proj/node6.html#SECTION0242000000000000000000>)" by Zdeněk Dvořák *et al.*

Examples

- http://www.aisee.com/graph_of_the_month/cfg.htm (http://www.aisee.com/graph_of_the_month/cfg.htm)
- <http://www.absint.com/aicall/gallery.htm> (<http://www.absint.com/aicall/gallery.htm>)
- <http://www.icd.de/es/icd-c/example.html> (<http://www.icd.de/es/icd-c/example.html>)
- <http://compilers.cs.ucla.edu/avrora/cfg.html> (<http://compilers.cs.ucla.edu/avrora/cfg.html>)

Basic block

In computing, a **basic block**^[1] is a portion of the code within a program with certain desirable properties that make it highly amenable to analysis. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Basic blocks form the vertices or nodes in a control flow graph.

Definition

The code in a basic block has:

- one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program;
- one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

Under these circumstances, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order.^[2]

The code may be source code, assembly code or some other sequence of instructions.

More formally, a sequence of instructions forms a basic block if:

- the instruction in each position dominates, or always executes before, all those in later positions, and
- no other instruction executes between two instructions in the sequence.

This definition is more general than the intuitive one in some ways. For example, it allows unconditional jumps to labels not targeted by other jumps. This definition embodies the properties that make basic blocks easy to work with when constructing an algorithm.

The blocks to which control may transfer after reaching the end of a block are called that block's *successors*, while the blocks from which control may have come when entering a block are called that block's *predecessors*. The start of a basic block may be jumped to from more than one location.

Algorithm to generate basic blocks.

The algorithm for generating basic blocks from a listing of code is simple: the analyser scans over the code, marking *block boundaries*, which are instructions which may either begin or end a block because they either transfer control or accept control from another point. Then, the listing is simply "cut" at each of these points, and basic blocks remain.

Note that this method does not always generate *maximal* basic blocks, by the formal definition, but they are usually sufficient (maximal basic blocks are basic blocks which cannot be extended by including adjacent blocks without violating the definition of a basic block^[3]).

[4] Input : A sequence of instructions (mostly Three address code).

Output: A list of basic blocks with each three-address statement in exactly one block.

Step 1. Identify the leaders in the code. Leaders are instructions which come under any of the following 3 categories :

1. The first instruction is a leader.
2. The target of a conditional or an unconditional goto/jump instruction is a leader.
3. The instruction that immediately follows a conditional or an unconditional goto/jump instruction is a leader.

Step 2. Starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader.

Thus every basic block has a leader.

Instructions that end a basic block include

- Unconditional and conditional branches, both direct and indirect
- Returns to a calling procedure
- Instructions which may throw an exception
- Function calls can be at the end of a basic block if they may not return, such as functions which throw exceptions or special calls like C's `long jmp` and `exit`
- The return instruction itself.

Instructions which begin a new basic block include

- Procedure and function entry points
- Targets of jumps or branches
- "Fall-through" instructions following some conditional branches
- Instructions following ones that throw exceptions
- Exception handlers.

Note that, because control can never pass through the end of a basic block, some block boundaries may have to be modified after finding the basic blocks. In particular, fall-through conditional branches must be changed to two-way branches, and function calls throwing exceptions must have unconditional jumps added after them. Doing these may require adding labels to the beginning of other blocks.

Notes

- [1] "Control Flow Analysis" by Frances E. Allen (<http://portal.acm.org/citation.cfm?id=808479>)
- [2] "Global Common Subexpression Elimination" by John Cocke (<http://portal.acm.org/citation.cfm?id=808480>)
- [3] Modern Compiler Design by Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen p320
- [4] Compiler Principles, Techniques and Tools, Aho Sethi Ullman

External links

- Basic Blocks - GNU Compiler Collection (<http://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html>)
- Extended Basic Block - Wiktionary (http://en.wiktionary.org/wiki/extended_basic_block)

Call graph

A **call graph** (also known as a call multigraph) is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g . Thus, a cycle in the graph indicates recursive procedure calls.

Call graphs are a basic program analysis result that can be used for human understanding of programs, or as a basis for further analyses, such as an analysis that tracks the flow of values between procedures. One simple application of call graphs is finding procedures that are never called.

Call graphs can be **dynamic** or **static**. A dynamic call graph is a record of an execution of the program, e.g., as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is undecidable, so static call graph algorithms are generally overapproximations. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Call graphs can be defined to represent varying degrees of precision. A more precise call graph more precisely approximates the behavior of the real program, at the cost of taking longer to compute and more memory to store. The most precise call graph is fully **context-sensitive**, which means that for each procedure, the graph contains a separate node for each call stack that procedure can be activated with. A fully context-sensitive call graph can be computed dynamically easily, although it may take up a large amount of memory. Fully context-sensitive call graphs are usually not computed statically, because it would take too long for a large program. The least precise call graph is **context-insensitive**, which means that there is only one node for each procedure.

With languages that feature dynamic dispatch, such as Java and C++, computing a static call graph precisely requires alias analysis results. Conversely, computing precise aliasing requires a call graph. Many static analysis systems solve the apparent infinite regress by computing both simultaneously.

This term is frequently used in the compiler and binary translation community. By tracking a call graph, it may be possible to detect anomalies of program execution or code injection attacks.

Software

Free software call-graph generators

cflow^[1]

GNU cflow is able to generate the direct and inverted call graph of a C program

KCachegrind^[2]

powerful tool to generate and analyze call graphs based on data generated by Valgrind's callgrind tool.

codeviz^[3]

a static call graph generator (the program is *not* run). Implemented as a patch to gcc; works for C and C++ programs.

egypt^[4]

a small Perl script that uses gcc and Graphviz to generate the static call graph of a C program.

Devel::NYTProf^[5]

a perl performance analyser and call chart generator

gprof

part of the GNU Binary Utilities

phpCallGraph^[6]

a call graph generator for PHP programs that uses Graphviz. It is written in PHP and requires at least PHP 5.2.

pycallgraph^[7]

a call graph generator for Python programs that uses Graphviz.

doxygen

Uses graphviz to generate static call/inheritance diagrams

Mac OS X Activity Monitor^[8]

Apple GUI process monitor Activity Monitor has a built-in call graph generator that can sample processes and return a call graph. This function is only available in Mac OS X Leopard

CCTree^[9]

Native Vim plugin that can display static call graphs by reading a cscope database. Works for C programs.

Proprietary call-graph generators

Project Analyzer

Static code analyzer and call graph generator for Visual Basic code

Intel VTune Performance Analyzer

Instrumenting profiler to show call graph and execution statistics

DMS Software Reengineering Toolkit

Customizable program analysis tool with static whole-program global call graph extraction for C, Java and COBOL

CodeProphet Profiler^[10]

Callgraph Profiler for native C/C++ code under Windows x86, x64 and Windows Mobile.

Other, related tools

Graphviz

Turns a text representation of any graph (including a call graph) into a picture. Must be used together with `gprof` (and a glue script, such as `gprof2dot`^[11]), which in accordance to the Unix philosophy doesn't handle graphics by itself.

Sample Graph

A sample Call Graph generated from `gprof` analyzing itself:

index	called	name	index	called	name
	72384/72384	sym_id_parse [54]		1508/1508	cg_dfn [15]
[3]	72384	match [3]	[13]	1508	pre_visit [13]
	4/9052	cg_tally [32]		1508/1508	cg_assemble [38]
	3016/9052	hist_print [49]	[14]	1508	propagate_time [14]
	6032/9052	propagate_flags [52]			
[4]	9052	sym_lookup [4]		2	cg_dfn [15]
				1507/1507	cg_assemble [38]
	5766/5766	core_create_function_syms [41] [15]		1507+2	cg_dfn [15]
[5]	5766	core_sym_class [5]		1509/1509	is_numbered [9]
				1508/1508	is_busy [11]
	24/1537	parse_spec [19]		1508/1508	pre_visit [13]
	1513/1537	core_create_function_syms [41]		1508/1508	post_visit [12]
[6]	1537	sym_init [6]		2	cg_dfn [15]
	1511/1511	core_create_function_syms [41]		1505/1505	hist_print [49]
[7]	1511	get_src_info [7]	[16]	1505	print_line [16]
				2/9	print_name_only [25]
	2/1510	arc_add [31]			
	1508/1510	cg_assemble [38]		1430/1430	core_create_function_syms [41]
[8]	1510	arc_lookup [8]	[17]	1430	source_file_lookup_path [17]
	1509/1509	cg_dfn [15]		24/24	sym_id_parse [54]
[9]	1509	is_numbered [9]	[18]	24	parse_id [18]
				24/24	parse_spec [19]
	1508/1508	propagate_flags [52]			
[10]	1508	inherit_flags [10]		24/24	parse_id [18]
			[19]	24	parse_spec [19]
	1508/1508	cg_dfn [15]		24/1537	sym_init [6]
[11]	1508	is_busy [11]			
				24/24	main [1210]
	1508/1508	cg_dfn [15]	[20]	24	sym_id_add [20]
[12]	1508	post_visit [12]			

References

- Ryder, B.G., "Constructing the Call Graph of a Program," Software Engineering, IEEE Transactions on , vol. SE-5, no.3pp. 216- 226, May 1979 [12]
- Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. SIGPLAN Not. 32, 10 (Oct. 1997), 108-124. [13]
- Callahan, D.; Carle, A.; Hall, M.W.; Kennedy, K., "Constructing the procedure call multigraph," Software Engineering, IEEE Transactions on , vol.16, no.4pp.483-487, Apr 1990 [14]

References

- [1] <http://www.gnu.org/software/cflow/>
- [2] <http://kcachegrind.sourceforge.net/cgi-bin/show.cgi>
- [3] <http://www.csn.ul.ie/~mel/projects/codeviz/>
- [4] <http://www.json.org/egypt/>
- [5] <http://search.cpan.org/perldoc?Devel::NYTProf>
- [6] <http://phpcallgraph.sourceforge.net/>
- [7] <http://pycallgraph.slowchop.com/>
- [8] <http://www.apple.com/>
- [9] http://www.vim.org/scripts/script.php?script_id=2368
- [10] <http://www.codeprophet.co.uk>
- [11] <http://code.google.com/p/google-gprof2dot/>
- [12] http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=35910&arnumber=1702621&count=17&index=5
- [13] <http://doi.acm.org/10.1145/263700.264352>
- [14] http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=1950&arnumber=54302&count=13&index=12

Data-flow analysis

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint. This general approach was developed by Gary Kildall while teaching at the Naval Postgraduate School.^[1]

Basic principles

It is the process of collecting information about the way the variables are used, defined in the program. Data-flow analysis attempts to obtain particular information at each point in a procedure. Usually, it is enough to obtain this information at the boundaries of basic blocks, since from that it is easy to compute the information at points in the basic block. In forward flow analysis, the exit state of a block is a function of the block's entry state. This function is the composition of the effects of the statements in the block. The entry state of a block is a function of the exit states of its predecessors. This yields a set of data-flow equations:

For each block b :

$$\begin{aligned} \text{out}_b &= \text{trans}_b(\text{in}_b) \\ \text{in}_b &= \text{join}_{p \in \text{pred}_b}(\text{out}_p) \end{aligned}$$

In this, trans_b is the **transfer function** of the block b . It works on the entry state in_b , yielding the exit state out_b . The join operation join combines the exit states of the predecessors $p \in \text{pred}_b$ of b , yielding the entry

state of b .

After solving this set of equations, the entry and / or exit states of the blocks can be used to derive properties of the program at the block boundaries. The transfer function of each statement separately can be applied to get information at a point inside a basic block.

Each particular type of data-flow analysis has its own specific transfer function and join operation. Some data-flow problems require backward flow analysis. This follows the same plan, except that the transfer function is applied to the exit state yielding the entry state, and the join operation works on the entry states of the successors to yield the exit state.

The entry point (in forward flow) plays an important role: Since it has no predecessors, its entry state is well defined at the start of the analysis. For instance, the set of local variables with known values is empty. If the control flow graph does not contain cycles (there were no explicit or implicit loops in the procedure) solving the equations is straightforward. The control flow graph can then be topologically sorted; running in the order of this sort, the entry states can be computed at the start of each block, since all predecessors of that block have already been processed, so their exit states are available. If the control flow graph does contain cycles, a more advanced algorithm is required.

An iterative algorithm

The most common way of solving the data-flow equations is by using an iterative algorithm. It starts with an approximation of the in-state of each block. The out-states are then computed by applying the transfer functions on the in-states. From these, the in-states are updated by applying the join operations. The latter two steps are repeated until we reach the so-called **fixpoint**: the situation in which the in-states (and the out-states in consequence) do not change.

A basic algorithm for solving data-flow equations is the **round-robin iterative algorithm**:

```

for  $i \leftarrow 1$  to  $N$ 
    initialize node  $i$ 
    while (sets are still changing)
        for  $i \leftarrow 1$  to  $N$ 
            recompute sets at node  $i$ 
```

Convergence

To be usable, the iterative approach should actually reach a fixpoint. This can be guaranteed by imposing constraints on the combination of the value domain of the states, the transfer functions and the join operation.

The value domain should be a partial order with **finite height** (i.e., there are no infinite ascending chains $x_1 < x_2 < \dots$). The combination of the transfer function and the join operation should be monotonic with respect to this partial order. Monotonicity ensures that on each iteration the value will either stay the same or will grow larger, while finite height ensures that it cannot grow indefinitely. Thus we will ultimately reach a situation where $T(x) = x$ for all x , which is the fixpoint.

The work list approach

It is easy to improve on the algorithm above by noticing that the in-state of a block will not change if the out-states of its predecessors don't change. Therefore, we introduce a **work list**: a list of blocks that still need to be processed. Whenever the out-state of a block changes, we add its successors to the work list. In each iteration, a block is removed from the work list. Its out-state is computed. If the out-state changed, the block's successors are added to the work list. For efficiency, a block should not be in the work list more than once.

The algorithm is started by putting the entry point in the work list. It terminates when the work list is empty.

The order matters

The efficiency of iteratively solving data-flow equations is influenced by the order at which local nodes are visited. Furthermore, it depends, whether the data-flow equations are used for forward or backward data-flow analysis over the CFG. Intuitively, in a forward flow problem, it would be fastest if all predecessors of a block have been processed before the block itself, since then the iteration will use the latest information. In the absence of loops it is possible to order the blocks in such a way that the correct out-states are computed by processing each block only once.

In the following, a few iteration orders for solving data-flow equations are discussed (a related concept to iteration order of a CFG is tree traversal of a tree).

- **Random order** - This iteration order is not aware whether the data-flow equations solve a forward or backward data-flow problem. Therefore, the performance is relatively poor compared to specialized iteration orders.
- **Postorder** - This is a typical iteration order for backward data-flow problems. In *postorder iteration*, a node is visited after all its successor nodes have been visited. Typically, the *postorder iteration* is implemented with the **depth-first** strategy.
- **Reverse postorder** - This is a typical iteration order for forward data-flow problems. In **reverse-postorder iteration**, a node is visited before all its successor nodes have been visited, except when the successor is reached by a back edge. (Note that this is not the same as preorder.)

Initialization

The initial value of the in-states is important to obtain correct and accurate results. If the results are used for compiler optimizations, they should provide **conservative** information, i.e. when applying the information, the program should not change semantics. The iteration of the fixpoint algorithm will take the values in the direction of the maximum element. Initializing all blocks with the maximum element is therefore not useful. At least one block starts in a state with a value less than the maximum. The details depend on the data-flow problem. If the minimum element represents totally conservative information, the results can be used safely even during the data-flow iteration. If it represents the most accurate information, fixpoint should be reached before the results can be applied.

Examples

The following are examples of properties of computer programs that can be calculated by data-flow analysis. Note that the properties calculated by data-flow analysis are typically only approximations of the real properties. This is because data-flow analysis operates on the syntactical structure of the CFG without simulating the exact control flow of the program. However, to be still useful in practice, a data-flow analysis algorithm is typically designed to calculate an upper respectively lower approximation of the real program properties.

Forward Analysis

The reaching definition analysis calculates for each program point the set of definitions that may potentially reach this program point.

```

1: if b==4 then
2:   a = 5;
3: else
4:   a = 3;
5: endif
6:
7: if a < 4 then
8:   ...

```

The reaching definition of variable "a" at line 7 is the set of assignments a=5 at line 2 and a=3 at line 4.

Backward Analysis

The live variable analysis calculates for each program point the variables that may be potentially read afterwards before their next write update. The result is typically used by dead code elimination to remove statements that assign to a variable whose value is not used afterwards.

The in-state of a block is the set of variables that are live at the end of the block. Its out-state is the set of variable that is live at the start of it. The in-state is the union of the out-states of the blocks successors. The transfer function of a statement is applied by making the variables that are written dead, then making the variables that are read live.

```
// out: {}
b1: a = 3;
    b = 5;
    d = 4;
    if a > b then
// in: {a,b,d}

// out: {a,b}
b2:     c = a + b;
        d = 2;
// in: {b,d}

// out: {b,d}
b3: endif
        c = 4;
        return b * d + c;
// in:{}
```

The out-state of b3 only contains *b* and *d*, since *c* has been written. The in-state of b1 is the union of the out-states of b2 and b3. The definition of *c* in b2 can be removed, since *c* is not live immediately after the statement.

Solving the data-flow equations starts with initializing all in-states and out-states to the empty set. The work list is initialized by inserting the exit point (b3) in the work list (typical for backward flow). Its computed out-state differs from the previous one, so its predecessors b1 and b2 are inserted and the process continues. The progress is summarized in the table below.

processing	in-state	old out-state	new out-state	work list
b3	{}	{}	{b,d}	(b1,b2)
b1	{b,d}	{}	{}	(b2)
b2	{b,d}	{}	{a,b}	(b1)
b1	{a,b,d}	{}	{}	0

Note that b1 was entered in the list before b2, which forced processing b1 twice (b1 was re-entered as predecessor of b2). Inserting b2 before b1 would have allowed earlier completion.

Initializing with the empty set is an optimistic initialization: all variables start out as dead. Note that the out-states cannot shrink from one iteration to the next, although the out-state can be smaller than the in-state. This can be seen from the fact that after the first iteration the out-state can only change by a change of the in-state. Since the in-state starts as the empty set, it can only grow in further iterations.

Other approaches

In 2002, Markus Mohnen described a new method of data-flow analysis that does not require the explicit construction of a data-flow graph,^[2] instead relying on abstract interpretation of the program and keeping a working set of program counters. At each conditional branch, both targets are added to the working set. Each path is followed for as many instructions as possible (until end of program or until it has looped with no changes), and then removed from the set and the next program counter retrieved.

Bit vector problems

The examples above are problems in which the data-flow value is a set, e.g. the set of reaching definitions (Using a bit for a definition position in the program), or the set of live variables. These sets can be represented efficiently as **bit vectors**, in which each bit represents set membership of one particular element. Using this representation, the join and transfer functions can be implemented as bitwise logical operations. The join operation is typically union or intersection, implemented by bitwise *logical or* and *logical and*. The transfer function for each block can be decomposed in so-called *gen* and *kill* sets.

As an example, in live-variable analysis, the join operation is union. The *kill* set is the set of variables that are written in a block, whereas the *gen* set is the set of variables that are read without being written first. The data-flow equations become

$$\begin{aligned} \text{out}_b &= \bigcup_{s \in \text{succ}_b} \text{in}_s \\ \text{in}_b &= (\text{out}_b - \text{kill}_b) \cup \text{gen}_b \end{aligned}$$

In logical operations, this reads as

$$\begin{aligned} \text{out}(b) &= 0 \\ \text{for } s \text{ in } \text{succ}(b) \\ \text{out}(b) &= \text{out}(b) \text{ or } \text{in}(s) \\ \text{in}(b) &= (\text{out}(b) \text{ and not } \text{kill}(b)) \text{ or } \text{gen}(b) \end{aligned}$$

Sensitivities

Data-flow analysis is inherently flow-sensitive. Data-flow analysis is typically path-insensitive, though it is possible to define data-flow equations that yield a path-sensitive analysis.

- A **flow-sensitive** analysis takes into account the order of statements in a program. For example, a flow-insensitive pointer alias analysis may determine "variables *x* and *y* may refer to the same location", while a flow-sensitive analysis may determine "after statement 20, variables *x* and *y* may refer to the same location".
- A **path-sensitive** analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions. For instance, if a branch contains a condition $x > 0$, then on the *fall-through* path, the analysis would assume that $x \leq 0$ and on the target of the branch it would assume that indeed $x > 0$ holds.
- A **context-sensitive** analysis is an *interprocedural* analysis that considers the calling context when analyzing the target of a function call. In particular, using context information one can *jump back* to the original call site, whereas without that information, the analysis information has to be propagated back to all possible call sites, potentially losing precision.

List of data-flow analyses

- Reaching definitions
- Liveness analysis
- Definite assignment analysis
- Available expression

Notes

- [1] Kildall, Gary (1973). "A Unified Approach to Global Program Optimization" (<http://portal.acm.org/citation.cfm?id=512945&coll=portal&dl=ACM>). *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*: 194–206. doi:10.1145/512927.512945. . Retrieved 2006-11-20.
- [2] Mohnen, Markus (2002). "A Graph-Free Approach to Data-Flow Analysis" (<http://www.springerlink.com/content/a57r0a6q999p15yc/>). *Lecture Notes in Computer Science* 2304: 185–213. doi:10.1007/3-540-45937-5_6. . Retrieved 2008-12-02.

Further reading

- Cooper, Keith D. and Torczon, Linda. *Engineering a Compiler*. Morgan Kaufmann. 2005.
- Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann. 1997.
- Hecht, Matthew S. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc. 1977.
- Khedker, Uday P. Sanyal, Amitabha Karkare, Bageshri. *Data Flow Analysis: Theory and Practice* (<http://www.cse.iitb.ac.in/~uday/dfaBook-web>), CRC Press (Taylor and Francis Group). 2009.
- Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer. 2005.

Use-define chain

A **Use-Definition Chain (UD Chain)** is a data structure that consists of a use, U, of a variable, and all the definitions, D, of that variable that can reach that use without any other intervening definitions. A definition can have many forms, but is generally taken to mean the assignment of some value to a variable (which is different from the use of the term that refers to the language construct involving a data type and allocating storage).

A counterpart of a *UD Chain* is a **Definition-Use Chain (DU Chain)**, which consists of a definition, D, of a variable and all the uses, U, reachable from that definition without any other intervening definitions.

Both UD and DU chains are created by using a form of static code analysis known as data flow analysis. Knowing the use-def and def-use chains for a program or subprogram is a prerequisite for many compiler optimizations, including constant propagation and common subexpression elimination.

Purpose

Making the use-define or define-use chains is a step in liveness analysis, so that logical representations of all the variables can be identified and tracked through the code.

Consider the following snippet of code:

```
int x = 0;      /* A */
x = x + y;    /* B */
/* 1, some uses of x */
x = 35;        /* C */
/* 2, some more uses of x */
```

Notice that `x` is assigned a value at three points (marked A, B, and C). However, at the point marked "1", the use-def chain for `x` should indicate that its current value must have come from line B (and its value at line B must

have come from line A). Contrariwise, at the point marked "2", the use-def chain for x indicates that its current value must have come from line C. Since the value of the x in block 2 does not depend on any definitions in block 1 or earlier, x might as well be a different variable there; practically speaking, it *is* a different variable — call it x_2 .

```
int x = 0;      /* A */
x = x + y;    /* B */
/* 1, some uses of x */
int x2 = 35;   /* C */
/* 2, some uses of x2 */
```

The process of splitting x into two separate variables is called live range splitting. See also static single assignment form.

Setup

The list of statements determines a strong order among statements.

- Statements are labeled using the following conventions: $s(i)$, where i is an integer in $[1, n]$; and n is the number of statements in the basic block
- Variables are identified in italic (e.g., v, u and t)
- Every variable is assumed to have a definition in the context or scope. (In static single assignment form, use-define chains are explicit because each chain contains a single element.)

For a variable, such as v , its declaration is identified as V (italic capital letter), and for short, its declaration is identified as $s(0)$. In general, a declaration of a variable can be in an outer scope (e.g., a global variable).

Definition of a Variable

When a variable, v , is on the LHS of an assignment statement, such as $s(j)$, then $s(j)$ is a definition of v . Every variable (v) has at least one definition by its declaration (V) (or initialization).

Use of a Variable

If variable, v , is on the RHS of statement $s(j)$, there is a statement, $s(i)$ with $i < j$ and $\min(j-i)$, that it is a definition of v and it has a use at $s(j)$ (or, in short, when a variable, v , is on the RHS of a statement $s(j)$, then v has a use at statement $s(j)$).

Execution

Consider the sequential execution of the list of statements, $s(i)$, and what can now be observed as the computation at statement, j :

- A definition at statement $s(i)$ with $i < j$ is **alive** at j , if it has a use at a statement $s(k)$ with $k \geq j$. The set of alive definitions at statement i is denoted as $A(i)$ and the number of alive definitions as $|A(i)|$. ($A(i)$ is a simple but powerful concept: theoretical and practical results in space complexity theory, access complexity(I/O complexity), register allocation and cache locality exploitation are based on $A(i)$.)
- A definition at statement $s(i)$ **kills** all previous definitions ($s(k)$ with $k < i$) for the same variables.

Execution example for def-use-chain

This example is based on a java algorithm for finding the ggt (it is not important to understand, what function the ggt / this code represents)

```
int ggt(int a, int b) {
    int c = a;
    int d = b;
    if(c == 0)
        return d;
    while(d != 0) {
        if(c > d)
            c = c - d;
        else
            d = d - c;
    }
    return c;
}
```

To find out all def-use-chains for variable d, do the following steps:

1. Search for the first time, the variable is defined (write access).

In this case it is "d=b" (l.3)

2. Search for the first time, the variable is read.

In this case it is "return d"

3. Write down this information in the following style:

[name of the variable you are creating a def-use-chain for, the concrete write access, the concrete read access]

In this case it is:

[d, d=b, return d]

Repeat this steps in the following style:

Combine each write access with each read access (but NOT the other way round)

The result should be:

1. [d, d=b, return d]
2. [d, d=b, while(d!=0)]
3. [d, d=b, if(c>d)]
4. [d, d=b, c=c-d]
5. [d, d=b, d=d-c]
6. [d, d=d-c, while(d!=0)]
7. [d, d=d-c, if(c>d)]
8. [d, d=d-c, c=c-d]
9. [d, d=d-c, d=d-c]

You have to take care, if the variable is changed by the time.

For example: From line 3 down to line 9, "d" is not redefined / changed.

At line 10, "d" could be redefined, this is, why you have to recombine this write access on "d" with all possible read access, which could be reached.

In this case, only the code beyond line 6 is relevant. Line 3 for example cannot be reached again.

For your understanding, you can imagine 2 different variables "d":

1. [d1, d1=b, return d1]

2. [d1, d1=b, while(d1!=0)]
3. [d1, d1=b, if(c>d1)]
4. [d1, d1=b, c=c-d1]
5. [d1, d1=b, d1=d1-c]
6. [d2, d2=d2-c, while(d2!=0)]
7. [d2, d2=d2-c, if(c>d2)]
8. [d2, d2=d2-c, c=c-d2]
9. [d2, d2=d2-c, d2=d2-c]

Method of building a *use-def* (or *ud*) chain

1. Set definitions in statement s(0)
2. For each i in [1,n], find live definitions that have use in statement s(i)
3. Make a link among definitions and uses
4. Set the statement s(i), as definition statement
5. Kill previous definitions

With this algorithm, two things are accomplished:

1. A directed acyclic graph (DAG) is created on the variable uses and definitions. The DAG specifies a data dependency among assignment statements, as well as a partial order (therefore parallelism among statements).
2. When statement $s(i)$ is reached, there is a list of *live* variable assignments. If only one assignment is live, for example, constant propagation might be used.

Live variable analysis

In compiler theory, **live variable analysis** (or simply **liveness analysis**) is a classic data flow analysis performed by compilers to calculate for each program point the variables that may be potentially read before their next write, that is, the variables that are *live* at the exit from each program point.

Stated simply: a variable is **live** if it holds a value that may be needed in the future.

It is a "backwards may" analysis. The analysis is done in a backwards order, and the dataflow confluence operator is set union.

```
L1: b := 3;
L2: c := 5;
L3: a := f(b + c);
      goto L1;
```

The set of live variables at line L3 is $\{b, c\}$ because both are used in the addition, and thereby the call to f and assignment to a . But the set of live variables at line L1 is only $\{b\}$ since variable c is updated in line 2. The value of variable a is never used, so the variable is never live. Note that f may be stateful, so the never-live assignment to a can be eliminated, but there is insufficient information to rule on the entirety of L3.

The dataflow equations used for a given basic block s and exiting block f in live variable analysis are (GEN(s) means the set of variables used in s before any assignment; KILL(s) means the set of variables assigned a value in s (in many books, KILL(s) is also defined as the set of variables assigned a value in s *before any use*, but this doesn't change the solution of the dataflow equation):

$$\text{LIVE}_{in}[s] = \text{GEN}[s] \cup (\text{LIVE}_{out}[s] - \text{KILL}[s])$$

$$\text{LIVE}_{out}[final] = \emptyset$$

$$\text{LIVE}_{out}[s] = \bigcup_{p \in \text{succ}[S]} \text{LIVE}_{in}[p]$$

$$\text{GEN}[d : y \leftarrow f(x_1, \dots, x_n)] = \{x_1, \dots, x_n\}$$

$$\text{KILL}[d : y \leftarrow f(x_1, \dots, x_n)] = \{y\}$$

The in-state of a block is the set of variables that are live at the start of the block. Its out-state is the set of variables that are live at the end of it. The out-state is the union of the in-states of the block's successors. The transfer function of a statement is applied by making the variables that are written dead, then making the variables that are read live.

```
// in: {}
b1: a = 3;
    b = 5;
    d = 4;
    x = 100; //x is never being used later thus not in the out set {a,b,d}
    if a > b then
// out: {a,b,d}      //union of all (in) successors of b1 => b2: {a,b}, and b3:{b,d}

// in: {a,b}
b2: c = a + b;
    d = 2;
// out: {b,d}

// in: {b,d}
b3: endif
    c = 4;
    return b * d + c;
// out:{}
```

The in-state of b3 only contains b and d , since c has been written. The out-state of b1 is the union of the in-states of b2 and b3. The definition of c in b2 can be removed, since c is not live immediately after the statement.

Solving the data flow equations starts with initializing all in-states and out-states to the empty set. The work list is initialized by inserting the exit point (b3) in the work list (typical for backward flow). Its computed out-state differs from the previous one, so its predecessors b1 and b2 are inserted and the process continues. The progress is summarized in the table below.

processing	in-state	old out-state	new out-state	work list
b3	{}	{}	{b,d}	(b1,b2)
b1	{b,d}	{}	{}	(b2)
b2	{a,b}	{}	{a,b}	(b1)
b1	{a,b,d}	{}	{}	0

Note that b1 was entered in the list before b2, which forced processing b1 twice (b1 was re-entered as predecessor of b2). Inserting b2 before b1 would have allowed earlier completion.

Initializing with the empty set is an optimistic initialization: all variables start out as dead. Note that the out-states cannot shrink from one iteration to the next, although the out-state can be smaller than the in-state. This can be seen from the fact that after the first iteration the out-state can only change by a change of the in-state. Since the in-state starts as the empty set, it can only grow in further iterations.

Recently as of 2006, various program analyses such as live variable analysis have been solved using Datalog. The Datalog specifications for such analyses are generally an order of magnitude shorter than their imperative counterparts (e.g. iterative analysis), and are at least as efficient.^[1]

References

- [1] Whaley et al. (2004). *Using Datalog with Binary Decision Diagrams for Program Analysis*.

Reaching definition

In compiler theory, a **reaching definition** for a given instruction is another instruction, the target variable of which may reach the given instruction without an intervening assignment. For example, in the following code:

```
d1 : y := 3
d2 : x := y
```

`d1` is a reaching definition at `d2`. In the following, example, however:

```
d1 : y := 3
d2 : y := 4
d3 : x := y
```

`d1` is no longer a reaching definition at `d3`, because `d2` kills its reach.

As analysis

The similarly named **reaching definitions** is a data-flow analysis which statically determines which definitions may reach a given point in the code. Because of its simplicity, it is often used as the canonical example of a data-flow analysis in textbooks. The data-flow confluence operator used is set union, and the analysis is forward flow. Reaching definitions are used to compute use-def chains and def-use chains.

The data-flow equations used for a given basic block S in reaching definitions are:

- $\text{REACH}_{\text{in}}[S] = \bigcup_{p \in \text{pred}[S]} \text{REACH}_{\text{out}}[p]$
- $\text{REACH}_{\text{out}}[S] = \text{GEN}[S] \cup (\text{REACH}_{\text{in}}[S] - \text{KILL}[S])$

In other words, the set of reaching definitions going into S are all of the reaching definitions from S 's predecessors, $\text{pred}[S]$. $\text{pred}[S]$ consists of all of the basic blocks that come before S in the control flow graph.

The reaching definitions coming out of S are all reaching definitions of its predecessors minus those reaching definitions whose variable is killed by S plus any new definitions generated within S .

For a generic instruction, we define the **GEN** and **KILL** sets as follows:

- $\text{GEN}[d : y \leftarrow f(x_1, \dots, x_n)] = \{d\}$
- $\text{KILL}[d : y \leftarrow f(x_1, \dots, x_n)] = \text{DEFS}[y] - \{d\}$

where $\text{DEFS}[y]$ is the set of all definitions that assign to the variable y . Here d is a unique label attached to the assigning instruction; thus, the domain of values in reaching definitions are these instruction labels.

Further reading

- Aho, Alfred V.; Sethi, Ravi; & Ullman, Jeffrey D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison Wesley. ISBN 0-201-10088-6.
- Appel, Andrew W. (1999). *Modern Compiler Implementation in ML*. Cambridge University Press. ISBN 0-521-58274-1.
- Cooper, Keith D.; & Torczon, Linda. (2005). *Engineering a Compiler*. Morgan Kaufmann. ISBN 1-55860-698-X.
- Muchnick, Steven S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann. ISBN 1-55860-320-4.

Three address code

In computer science, **three-address code** (often abbreviated to TAC or 3AC) is a form of representing intermediate code used by compilers to aid in the implementation of code-improving transformations. Each instruction in three-address code can be described as a 4-tuple: (operator, operand1, operand2, result).

Each statement has the general form of:

$$\text{result} := \text{operand}_1 \text{ operator } \text{operand}_2$$

such as:

$$x := y \text{ op } z$$

where x , y and z are variables, constants or temporary variables generated by the compiler. op represents any operator, e.g. an arithmetic operator.

Expressions containing more than one fundamental operation, such as:

$$p := x + y \times z$$

are not representable in three-address code as a single instruction. Instead, they are decomposed into an equivalent series of instructions, such as

$$\begin{aligned} t_1 &:= y \times z \\ p &:= x + t_1 \end{aligned}$$

The term *three-address code* is still used even if some instructions use more or fewer than two operands. The key features of three-address code are that every instruction implements exactly one fundamental operation, and that the source and destination may refer to any available register.

A refinement of three-address code is static single assignment form (SSA).

Example

```
int main(void)
{
    int i;
    int b[10];
    for (i = 0; i < 10; ++i) {
        b[i] = i*i;
    }
}
```

The preceding C program, translated into three-address code, might look something like the following:

```

        i := 0           ; assignment
L1:   if i >= 10 goto L2      ; conditional jump
      t0 := i*i
      t1 := &b          ; address-of operation
      t2 := t1 + i      ; t2 holds the address of b[i]
      *t2 := t0         ; store through pointer
      i := i + 1
      goto L1

L2:

```

External links

- CSc 453: A Three-Address Intermediate Code Instruction Set for C-- ^[1]

References

[1] <http://www.cs.arizona.edu/classes/cs453/fall07/DOCS/intcode.html>

Static single assignment form

In compiler design, **static single assignment form** (often abbreviated as **SSA form** or simply **SSA**) is a property of an intermediate representation (IR), which says that each variable is assigned exactly once. Existing variables in the original IR are split into *versions*, new variables typically indicated by the original name with a subscript in textbooks, so that every definition gets its own version. In SSA form, use-def chains are explicit and each contains a single element.

SSA was developed by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, researchers at IBM in the 1980s.

In functional language compilers, such as those for Scheme, ML and Haskell, continuation-passing style (CPS) is generally used while one might expect to find SSA in a compiler for Fortran or C. SSA is formally equivalent to a well-behaved subset of CPS (excluding non-local control flow, which does not occur when CPS is used as intermediate representation), so optimizations and transformations formulated in terms of one immediately apply to the other.

Benefits

The primary usefulness of SSA comes from how it simultaneously simplifies and improves the results of a variety of compiler optimizations, by simplifying the properties of variables. For example, consider this piece of code:

```

y := 1
y := 2
x := y

```

Humans can see that the first assignment is not necessary, and that the value of *y* being used in the third line comes from the second assignment of *y*. A program would have to perform reaching definition analysis to determine this. But if the program is in SSA form, both of these are immediate:

```

y1 := 1
y2 := 2
x1 := y2

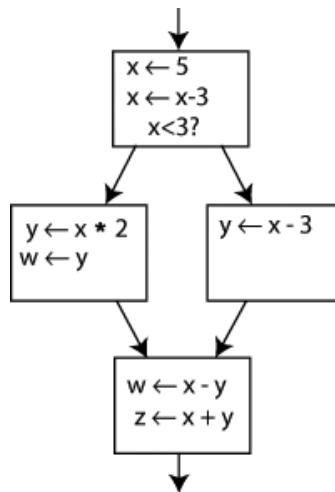
```

Compiler optimization algorithms which are either enabled or strongly enhanced by the use of SSA include:

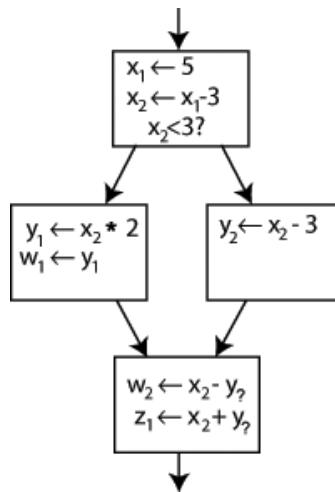
- constant propagation
- sparse conditional constant propagation
- dead code elimination
- global value numbering
- partial redundancy elimination
- strength reduction
- register allocation

Converting to SSA

Converting ordinary code into SSA form is primarily a simple matter of replacing the target of each assignment with a new variable, and replacing each use of a variable with the "version" of the variable reaching that point. For example, consider the following control flow graph:

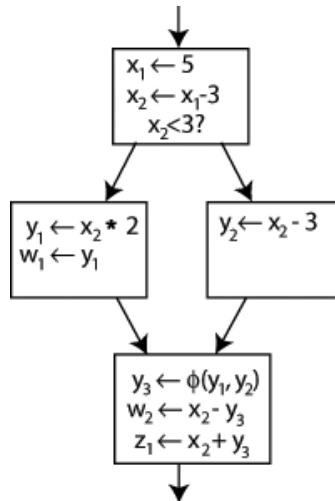


Notice that we could change the name on the left side of " $x \leftarrow x - 3$ ", and change the following uses of x to use that new name, and the program would still do the same thing. We exploit this in SSA by creating two new variables, x_1 and x_2 , each of which is assigned only once. We likewise give distinguishing subscripts to all the other variables, and we get this:



We've figured out which definition each use is referring to, except for one thing: the uses of y in the bottom block could be referring to *either* y_1 or y_2 , depending on which way the control flow came from. So how do we know which one to use?

The answer is that we add a special statement, called a Φ (*Phi*) function, to the beginning of the last block. This statement will generate a new definition of y , y_3 , by "choosing" either y_1 or y_2 , depending on which arrow control arrived from:



Now, the uses of y in the last block can simply use y_3 , and they'll obtain the correct value either way. You might ask at this point, do we need to add a Φ function for x too? The answer is no; only one version of x , namely x_2 is reaching this place, so there's no problem.

A more general question along the same lines is, given an arbitrary control flow graph, how can I tell where to insert Φ functions, and for what variables? This is a difficult question, but one that has an efficient solution that can be computed using a concept called *dominance frontiers*.

Note: the Φ functions are not actually implemented; instead, they're just markers for the compiler to place the value of all the variables, which are grouped together by the Φ function, in the same location in memory (or same register). According to Kenny Zadeck [1] Φ functions were originally known as *phoney* functions while SSA was being developed at IBM Research in the 1980s. The formal name of Φ function was only adopted when the work was first published in an academic paper.

Computing minimal SSA using dominance frontiers

First, we need the concept of a *dominator*: we say that a node A *strictly dominates* a different node B in the control flow graph if it's impossible to reach B without passing through A first. This is useful, because if we ever reach B we know that any code in A has run. We say that A *dominates* B (B is *dominated by* A) if either A strictly dominates B or A = B.

Now we can define the *dominance frontier*: a node B is in the dominance frontier of a node A if A does *not* strictly dominate B, but does dominate some immediate predecessor of B (possibly node A is the immediate predecessor of B. Then, because any node dominates itself and node A dominates itself, node B is in the dominance frontier of node A.). From A's point of view, these are the nodes at which other control paths, which don't go through A, make their earliest appearance.

Dominance frontiers capture the precise places at which we need Φ functions: if the node A defines a certain variable, then that definition and that definition alone (or redefinitions) will reach every node A dominates. Only when we leave these nodes and enter the dominance frontier must we account for other flows bringing in other definitions of the same variable. Moreover, no other Φ functions are needed in the control flow graph to deal with A's definitions, and we can do with no less.

One algorithm for computing the dominance frontier set^[2] is:

```

for each node b
  if the number of immediate predecessors of b ≥ 2
  
```

```

for each p in immediate predecessors of b
    runner := p
    while runner ≠ doms(b)
        add b to runner's dominance frontier set
        runner := doms(runner)

```

Note: in the code above, an immediate predecessor of node n is any node from which control is transferred to node n, and doms(b) is the node that immediately dominates node b (a singleton set).

There is an efficient algorithm for finding dominance frontiers of each node. This algorithm was originally described in Cytron *et al.* 1991.^[3] Also useful is chapter 19 of the book "Modern compiler implementation in Java" by Andrew Appel (Cambridge University Press, 2002). See the paper for more details.

Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy of Rice University describe an algorithm in their paper titled *A Simple, Fast Dominance Algorithm*.^[2] The algorithm uses well-engineered data structures to improve performance.

Variations that reduce the number of Φ functions

"Minimal" SSA inserts the minimal number of Φ functions required to ensure that each name is assigned a value exactly once and that each reference (use) of a name in the original program can still refer to a unique name. (The latter requirement is needed to ensure that the compiler can write down a name for each operand in each operation.) However, some of these Φ functions could be *dead*. For this reason, minimal SSA does not necessarily produce the fewest number of Φ functions that are needed by a specific procedure. For some types of analysis, these Φ functions are superfluous and can cause the analysis to run less efficiently.

Pruned SSA

Pruned SSA form is based on a simple observation: Φ functions are only needed for variables that are "live" after the Φ function. (Here, "live" means that the value is used along some path that begins at the Φ function in question.) If a variable is not live, the result of the Φ function cannot be used and the assignment by the Φ function is dead.

Construction of pruned SSA form uses live variable information in the Φ function insertion phase to decide whether a given Φ function is needed. If the original variable name isn't live at the Φ function insertion point, the Φ function isn't inserted.

Another possibility is to treat pruning as a dead code elimination problem. Then, a Φ function is live only if any use in the input program will be rewritten to it, or if it will be used as an argument in another Φ function. When entering SSA form, each use is rewritten to the nearest definition that dominates it. A Φ function will then be considered live as long as it is the nearest definition that dominates at least one use, or at least one argument of a live Φ .

Semi-pruned SSA

Semi-pruned SSA form^[4] is an attempt to reduce the number of Φ functions without incurring the relatively high cost of computing live variable information. It is based on the following observation: if a variable is never live upon entry into a basic block, it never needs a Φ function. During SSA construction, Φ functions for any "block-local" variables are omitted.

Computing the set of block-local variables is a simpler and faster procedure than full live variable analysis, making semi-pruned SSA form more efficient to compute than pruned SSA form. On the other hand, semi-pruned SSA form will contain more Φ functions.

Converting out of SSA form

As SSA form is no longer useful for direct execution, it is frequently used "on top of" another IR with which it remains in direct correspondence. This can be accomplished by "constructing" SSA as a set of functions which map between parts of the existing IR (basic blocks, instructions, operands, *etc.*) and its SSA counterpart. When the SSA form is no longer needed, these mapping functions may be discarded, leaving only the now-optimized IR.

Performing optimizations on SSA form usually leads to entangled SSA-Webs, meaning there are phi instructions whose operands do not all have the same root operand. In such cases color-out algorithms are used to come out of SSA. Naive algorithms introduce a copy along each predecessor path which caused a source of different root symbol to be put in phi than the destination of phi. There are multiple algorithms for coming out of SSA with fewer copies, most use interference graphs or some approximation of it to do copy coalescing.

Extensions

Extensions to SSA form can be divided into two categories.

Renaming scheme extensions alter the renaming criterion. Recall that SSA form renames each variable when it is assigned a value. Alternative schemes include static single use form (which renames each variable at each statement when it is used) and static single information form (which renames each variable when it is assigned a value, and at the post-dominance frontier).

Feature-specific extensions retain the single assignment property for variables, but incorporate new semantics to model additional features. Some feature-specific extensions model high-level programming language features like arrays, objects and aliased pointers. Other feature-specific extensions model low-level architectural features like speculation and predication.

Compilers using SSA form

SSA form is a relatively recent development in the compiler community. As such, many older compilers only use SSA form for some part of the compilation or optimization process, but most do not rely on it. Examples of compilers that rely heavily on SSA form include:

- The ETH Oberon-2 compiler was one of the first public projects to incorporate "GSA", a variant of SSA.
- The LLVM Compiler Infrastructure uses SSA form for all scalar register values (everything except memory) in its primary code representation. SSA form is only eliminated once register allocation occurs, late in the compile process (often at link time).
- The Open64 compiler uses SSA form in its global scalar optimizer, though the code is brought into SSA form before and taken out of SSA form afterwards. Open64 uses extensions to SSA form to represent memory in SSA form as well as scalar values.
- As of version 4 (released in April 2005) GCC, the GNU Compiler Collection, makes extensive use of SSA. The frontends generate "GENERIC" code which is then converted into "GIMPLE" code by the "gimplifier". High-level optimizations are then applied on the SSA form of "GIMPLE". The resulting optimized intermediate code is then translated into RTL, on which low-level optimizations are applied. The architecture-specific backends finally turn RTL into assembly language.
- IBM's open source adaptive Java virtual machine, Jikes RVM, uses extended Array SSA, an extension of SSA that allows analysis of scalars, arrays, and object fields in a unified framework. Extended Array SSA analysis is only enabled at the maximum optimization level, which is applied to the most frequently executed portions of code.
- In 2002, researchers modified^[5] IBM's JikesRVM (named Jalapeño at the time) to run both standard Java byte-code and a typesafe SSA (SafeTSA) byte-code class files, and demonstrated significant performance benefits

to using the SSA byte-code.

- Sun Microsystems' Java HotSpot Virtual Machine uses an SSA-based intermediate language in its JIT compiler.
- Mono uses SSA in its JIT compiler called Mini.
- jackcc^[6] is an open-source compiler for the academic instruction set Jackal 3.0. It uses a simple 3-operand code with SSA for its intermediate representation. As an interesting variant, it replaces Φ functions with a so-called SAME instruction, which instructs the register allocator to place the two live ranges into the same physical register.
- Although not a compiler, the Boomerang^[7] decompiler uses SSA form in its internal representation. SSA is used to simplify expression propagation, identifying parameters and returns, preservation analysis, and more.
- Portable.NET uses SSA in its JIT compiler.
- libFirm^[8] a completely graph based SSA intermediate representation for compilers. libFirm uses SSA form for all scalar register values until code generation by use of an SSA-aware register allocator.
- The Illinois Concert Compiler circa 1994 [9] used a variant of SSA called SSU (Static Single Use) which renames each variable when it is assigned a value, and in each conditional context in which that variable is used; essentially the static single information form mentioned above. The SSU form is documented in John Plevyak's Ph.D Thesis^[10].
- The COINS compiler uses SSA form optimizations as explained here: <http://www.is.titech.ac.jp/~sassa/coins-www-ssa/english/>
- The Chromium V8 JavaScript engine implements SSA in its Crankshaft compiler infrastructure as announced in December 2010^[11]
- PyPy uses a linear SSA representation for traces in its JIT compiler.
- Android's Dalvik virtual machine uses SSA in its JIT compiler.
- The Standard ML compiler MLton uses SSA in one of its intermediate languages.

References

Notes

- [1] Zadeck, F. Kenneth, *Presentation on the History of SSA* at the SSA'09 Seminar (<http://www.prog.uni-saarland.de/ssasem/>), Autrans, France, April 2009
- [2] Cooper, Keith D.; Harvey, Timothy J.; and Kennedy, Ken (2001). *A Simple, Fast Dominance Algorithm* (<http://www.cs.rice.edu/~keith/EMBED/dom.pdf>) .
- [3] Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N.; and Zadeck, F. Kenneth (1991). "Efficiently computing static single assignment form and the control dependence graph" (http://www.eecs.umich.edu/~mahlke/583w03/reading/cytron_toplas_91.pdf). *ACM Transactions on Programming Languages and Systems* **13** (4): 451–490. doi:10.1145/115372.115320. .
- [4] Briggs, Preston; Cooper, Keith D.; Harvey, Timothy J.; and Simpson, L. Taylor (1998). *Practical Improvements to the Construction and Destruction of Static Single Assignment Form* (http://www.cs.rice.edu/~harv/my_papers/ssa.pdf). .
- [5] <http://citeseer.ist.psu.edu/721276.html>
- [6] <http://jackcc.sf.net>
- [7] <http://boomerang.sourceforge.net/>
- [8] <http://www.info.uni-karlsruhe.de/software/libfirm>
- [9] <http://www-csag.ucsd.edu/projects/concert.html>
- [10] <http://www-csag.ucsd.edu/papers/jplevyak-thesis.ps>
- [11] <http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html>

General references

- Appel, Andrew W. (1999). *Modern Compiler Implementation in ML*. Cambridge University Press. ISBN 0-521-58274-1. Also available in Java (ISBN 0-521-82060-X 2002) and C (ISBN 0-521-60765-5, 1998) versions.
- Cooper, Keith D.; and Torczon, Linda (2003). *Engineering a Compiler*. Morgan Kaufmann. ISBN 1-55860-698-X.
- Muchnick, Steven S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann. ISBN 1-55860-320-4.
- Kelsey, Richard A. (March 1995). "A Correspondence between Continuation Passing Style and Static Single Assignment Form". *ACM SIGPLAN Notices* **30** (3): 13–22. doi:10.1145/202530.202532.
- Appel, Andrew W. (April 1998). "SSA is Functional Programming". *ACM SIGPLAN Notices* **33** (4): 17–20. doi:10.1145/278283.278285.
- Pop, Sebastian (2006). *The SSA Representation Framework: Semantics, Analyses and GCC Implementation* (<http://cri.ensmp.fr/people/pop/papers/2006-12-thesis.pdf>).

External links

- Bosscher, Steven; and Novillo, Diego. GCC gets a new Optimizer Framework (<http://lwn.net/Articles/84888/>). An article about GCC's use of SSA and how it improves over older IRs.
- The SSA Bibliography (<http://www.cs.man.ac.uk/~jsinger/ssa.html>). Extensive catalogue of SSA research papers.
- Zadeck, F. Kenneth. "The Development of Static Single Assignment Form" (<http://webcast.rice.edu/webcast.php?action=details&event=1346>), December 2007 talk on the origins of SSA.

Dominator

In computer science, in control flow graphs, a node **d** *dominates* a node **n** if every path from the *start node* to **n** must go through **d**. Notationally, this is written as **d** dom **n** (or sometimes **d** ≫ **n**). By definition, every node dominates itself.

There are a number of related concepts:

- A node **d** *strictly dominates* a node **n** if **d** dominates **n** and **d** does not equal **n**.
- The *immediate dominator* or **idom** of a node **n** is the unique node that strictly dominates **n** but does not strictly dominate any other node that strictly dominates **n**. Not all nodes have immediate dominators (e.g. entry nodes).
- The *dominance frontier* of a node **d** is the set of all nodes **n** such that **d** dominates an immediate predecessor of **n**, but **d** does not strictly dominate **n**. It is the set of nodes where **d**'s dominance stops.
- A *dominator tree* is a tree where each node's children are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree. The start node is the root of the tree.

History

Dominance was first introduced by Reese T. Prosser in a 1959 paper on analysis of flow diagrams.^[1] Prosser did not present an algorithm for computing dominance, which had to wait ten years for Edward S. Lowry and C. W. Medlock.^[2] Ron Cytron *et al.* rekindled interest in dominance in 1989 when they applied it to efficient computation of φ functions, which are used in static single assignment form.^[3]

Applications

Dominators, and dominance frontiers particularly, have applications in compilers for computing static single assignment form. A number of compiler optimizations can also benefit from dominators. The flow graph in this case comprises basic blocks.

Automatic parallelization benefits from postdominance frontiers. This is an efficient method of computing control dependence, which is critical to the analysis.

Memory usage analysis can benefit from the dominator tree to easily find leaks and identify high memory usage (<http://www.eclipse.org/mat/>).

Algorithms

The dominators of a node n are given by the maximal solution to the following data-flow equations:

$$\text{Dom}(n_o) = \{n_o\}$$

$$\text{Dom}(n) = \left(\bigcap_{p \in \text{preds}(n)} \text{Dom}(p) \right) \cup \{n\}$$

where n_o is the start node.

The dominator of the start node is the start node itself. The set of dominators for any other node n is the intersection of the set of dominators for all predecessors p of n . The node n is also in the set of dominators for n .

An algorithm for direct solution is:

```
// dominator of the start node is the start itself
Dom(n0) = {n0}
// for all other nodes, set all nodes as the dominators
for each n in N - {n0}
    Dom(n) = N;
// iteratively eliminate nodes that are not dominators
while changes in any Dom(n)
    for each n in N - {n0}:
        Dom(n) = {n} union with intersection over all p in pred(n) of Dom(p)
```

Direct solution is quadratic in the number of nodes, or $O(n^2)$. Lengauer and Tarjan developed an algorithm which is almost linear, but its implementation tends to be complex and time consuming for a graph of several hundred nodes or fewer.^[4]

Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy of Rice University describe an algorithm that essentially solves the above data flow equations but uses well engineered data structures to improve performance.^[5]

Postdominance

Analogous to the definition of dominance above, a node z is said to **post-dominate** a node n if all paths to the exit node of the graph starting at n must go through z . Similarly, the **immediate post-dominator** of a node n is the postdominator of n that doesn't strictly postdominate any other strict postdominators of n .

References

- [1] Prosser, Reese T. (1959). "Applications of Boolean matrices to the analysis of flow diagrams" (http://portal.acm.org/ft_gateway.cfm?id=1460314&type=pdf&coll=GUIDE&dl=GUIDE&CFID=79528182&CFTOKEN=33765747). *AFIPS Joint Computer Conferences: Papers presented at the December 1–3, 1959, eastern joint IRE-AIEE-ACM computer conference* (Boston, MA: ACM): 133–138. .
- [2] Lowry, Edward S.; and Medlock, Cleburne W. (January 1969). "Object code optimization" (http://portal.acm.org/ft_gateway.cfm?id=362838&type=pdf&coll=GUIDE&dl=GUIDE&CFID=79528182&CFTOKEN=33765747). *Communications of the ACM* **12** (1): 13–22. .
- [3] Cytron, Ron; Hind, Michael; and Hsieh, Wilson (1989). "Automatic Generation of DAG Parallelism" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.9287&rep=rep1&type=pdf>). *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*: 54–68. .
- [4] Lengauer, Thomas; and Tarjan, Robert Endre (July 1979). "A fast algorithm for finding dominators in a flowgraph" (http://portal.acm.org/ft_gateway.cfm?id=357071&type=pdf&coll=GUIDE&dl=GUIDE&CFID=79528182&CFTOKEN=33765747). *ACM Transactions on Programming Languages and Systems (TOPLAS)* **1** (1): 121–141. .
- [5] Cooper, Keith D.; Harvey, Timothy J; and Kennedy, Ken (2001). "A Simple, Fast Dominance Algorithm" (<http://www.hipersoft.rice.edu/grads/publications/dom14.pdf>). .

External links

- The Machine-SUIF Control Flow Analysis Library (<http://www.eecs.harvard.edu/hube/software/nci/cfa.html>)

C3 linearization

In computing, the **C3 superclass linearization** is an algorithm used primarily to obtain a consistent linearization of a multiple inheritance hierarchy in object-oriented programming. This linearization is used to resolve the order in which methods should be inherited, and is often termed "MRO" for Method Resolution Order. The name C3 refers to the three important properties of the resulting linearization: a consistent extended precedence graph, preservation of local precedence order, and monotonicity. It was first published at the 1996 OOPSLA conference, in a paper entitled "A Monotonic Superclass Linearization for Dylan".^[1] Subsequently, it has been chosen as the default algorithm for method resolution in Python 2.3 (and newer),^[2] ^[3] Perl 6,^[4] and Parrot.^[5] It is also available as an alternative, non-default MRO in the core of Perl 5 starting with version 5.10.0.^[6] An extension implementation for earlier versions of Perl 5 named `Class::C3` exists on CPAN.^[7]

References

- [1] "A Monotonic Superclass Linearization for Dylan" (<http://www.webcom.com/haahr/dylan/linearization-oopsla96.html>). *OOPSLA '96 Conference Proceedings*. ACM Press. 1996-06-28. pp. 69–82. doi:10.1145/236337.236343. ISBN 0-89791-788-X. .
- [2] Python 2.3's use of C3 MRO (<http://www.python.org/download/releases/2.3/mro/>)
- [3] Tutorial for practical applications of C3 linearization using Python (<http://rhettinger.wordpress.com/2011/05/26/super-considered-super/>)
- [4] Perl 6 will use C3 MRO (<http://use.perl.org/~autrijus/journal/25768>)
- [5] Parrot uses C3 MRO (<http://aspn.activestate.com/ASPN/Mail/Message/perl6-internals/2746631>)
- [6] C3 MRO available in Perl 5.10 (<http://search.cpan.org/dist/perl-5.10.0/lib/mro.pm>)
- [7] Perl 5 extension for C3 MRO on CPAN (<http://search.cpan.org/dist/Class-C3/>)

Intrinsic function

In compiler theory, an **intrinsic function** is a function available for use in a given language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the intrinsic function and can therefore better integrate it and optimize it for the situation. This is also called *builtin function* in many languages.

Compilers that implement intrinsic functions generally enable them only when the user has requested optimization, falling back to a default implementation provided by the language runtime environment otherwise.

Intrinsic functions are often used to explicitly implement vectorization and parallelization in languages which do not address such constructs. Altivec and OpenMP are examples of APIs which use intrinsic functions to declare, respectively, vectorizable and multiprocessor-aware operations during compilation. The compiler parses the intrinsic functions and converts them into vector math or multiprocessing object code appropriate for the target platform.

Microsoft and Intel's C/C++ compilers as well as GCC implement intrinsics that map directly to the x86 SIMD instructions (MMX, SSE, SSE2, SSE3, SSSE3, SSE4). In the latest version of the Microsoft compiler (VC2005 as well as VC2008) inline assembly is not available when compiling for 64 bit Windows.^[1] To compensate for the lack of inline assembly, new intrinsics have been added that map to standard assembly instructions that are not normally accessible through C/C++ (e.g.: bit scan).

References

- [1] Visual Studio 2010 SDK. "Intrinsics and Inline Assembly" (<http://msdn.microsoft.com/en-us/library/wbk4z78b.aspx?ppud=4>). Microsoft. . Retrieved 2010-04-16.

Aliasing

In computing, **aliasing** describes a situation in which a data location in memory can be accessed through different symbolic names in the program. Thus, modifying the data through one name implicitly modifies the values associated to all aliased names, which may not be expected by the programmer. As a result, aliasing makes it particularly difficult to understand, analyze and optimize programs. Aliasing analysers intend to make and compute useful information for understanding aliasing in programs.

Examples

Array bounds checking

For example, the C programming language does not perform array bounds checking. One can then exploit the implementation of the programming language by the compiler, plus the computer architecture's assembly language conventions, to achieve aliasing effects.

If an array is created on the stack, with a variable laid out in memory directly beside that array, one could index outside that array and then directly change that variable by changing the relevant array element. For example, if we have an `int` array of size ten (for this example's sake, calling it `vector`), next to another `int` variable (call it `i`), `vector[10]` (i.e. the 11th element) would be aliased to `i` if they are adjacent in memory.

This is possible in some implementations of C because an array is in reality a block of contiguous memory, and array elements are merely offsets off the address of the beginning of that block multiplied by the size of a single element. Since C has no bounds checking, indexing and addressing outside of the array is possible. Note that the aforementioned aliasing behaviour is implementation specific. Some implementations may leave space between arrays and variables on the stack, for instance, to align variables to memory locations that are a multiple of the architecture's native word size. The C standard does not generally specify how data is to be laid out in memory. (ISO/IEC 9899:1999, section 6.2.6.1).

It is not erroneous for a compiler to omit aliasing effects for accesses that fall outside the bounds of an array.

Aliased pointers

Another variety of aliasing can occur in any language that can refer to one location in memory with more than one name (for example, with pointers). See the C example of the xor swap algorithm that is a function; it assumes the two pointers passed to it are distinct, but if they are in fact equal (or aliases of each other), the function fails. This is a common problem with functions that accept pointer arguments, and their tolerance (or the lack thereof) for aliasing must be carefully documented, particularly for functions that perform complex manipulations on memory areas passed to them.

Specified aliasing

Controlled aliasing behaviour may be desirable in some cases (that is, aliasing behaviour that is specified, unlike that relevant to memory layout in C). It is common practice in Fortran. The Perl programming language specifies, in some constructs, aliasing behaviour, such as in `foreach` loops. This allows certain data structures to be modified directly with less code. For example,

```
my @array = (1, 2, 3);

foreach my $element (@array) {
    # Increment $element, thus automatically
```

```
# modifying @array, since $element is ''aliased''
# to each of @array's elements in turn.
$element++;
}

print "@array \n";
```

will print out "2 3 4" as a result. If one wanted to bypass aliasing effects, one could copy the contents of the index variable into another and change the copy.

Conflicts with optimization

Optimizers often have to make conservative assumptions about variables in the presence of pointers and references. For example, a constant propagation process that knows the value of variable `x` is 5 would not be able to keep using this information after an assignment to another variable (for example, `*y = 10`) because it could be that `*y` is an alias of `x`. This could be the case after an assignment like `y = &x`. In a function call arguments passed by reference or as pointers can have this same problem.

As an effect of the assignment to `*y`, the value of `x` would be changed as well, so propagating the information that `x` is 5 to the statements following `*y = 10` would be potentially wrong (if `*y` is indeed an alias of `x`). However, if we have information about pointers, the constant propagation process could make a query like: can `x` be an alias of `*y`? Then, if the answer is no, `x = 5` can be propagated safely.

Another optimization impacted by aliasing is code reordering. If the compiler decides that `x` is not an alias of `*y`, then code that uses or changes the value of `x` can be moved before the assignment `*y = 10`, if this would improve scheduling or enable more loop optimizations to be carried out.

To enable such optimizations in a predictable manner, the ISO standard for the C programming language (including its newer C99 edition, see section 6.5, paragraph 7) specifies that it is illegal (with some exceptions) for pointers or references of different types to reference the same memory location. This rule, known as "strict aliasing", sometimes allows for impressive increases in performance,^[1] but has been known to break some otherwise valid code. Several software projects intentionally violate this portion of the C99 standard. For example, Python 2.x did so to implement reference counting,^[2] and required changes to the basic object structs in Python 3 to enable this optimisation. The Linux kernel does this because strict aliasing causes problems with optimization of inlined code.^[3] In such cases, when compiled with `gcc`, the option `-fno-strict-aliasing` is invoked to prevent unwanted or invalid optimizations that could produce incorrect code. The C++ (programming language) has the same restrictions, and in fact, they are not in any sense new. As long as there have been standards for C and C++ there have been sections on aliasing in the standards. The only difference now is that compiler writers have gotten far enough along in the development of optimization techniques that developers who wrote code that broke with the standard in this way are startled and dismayed to find that they've been writing incorrect code.

External links

- Understanding Strict Aliasing ^[4] - Mike Acton
- Aliasing, pointer casts and gcc 3.3 ^[5] - Informational article on NetBSD mailing list
- Type-based alias analysis in C++ ^[6] - Informational article on type-based alias analysis in C++
- Understanding C/C++ Strict Aliasing ^[7]

Notes

- [1] Mike Acton (2006-06-01). "Understanding Strict Aliasing" (<http://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html>)..
- [2] Neil Schemenauer (2003-07-17). "ANSI strict aliasing and Python" (<http://mail.python.org/pipermail/python-dev/2003-July/036898.html>)..
- [3] Linus Torvalds (2003-02-26). "Re: Invalid compilation without -fno-strict-aliasing" (<http://lkml.org/lkml/2003/2/26/158>)..
- [4] <http://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html>
- [5] <http://mail-index.netbsd.org/tech-kern/2003/08/11/0001.html>
- [6] http://www.ddj.com/cpp/184404273;jsessionid=NV5BWY3EOHMFSQSNDLPC KH0CJUNN2JVN?_requestid=510121
- [7] <http://dbp-consulting.com/tutorials/StrictAliasing.html>

Alias analysis

Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location.

Alias analysis techniques are usually classified by flow-sensitivity and context-sensitivity. They may determine may-alias or must-alias information. The term **alias analysis** is often used interchangeably with term points-to analysis, a specific case.

What Does Alias Analysis Do?

In general, alias analysis determines whether or not separate memory references point to the same area of memory. This allows the compiler to determine what variables in the program will be affected by a statement. For example, consider the following section of code that accesses members of structures:

```
p.foo = 1;  
q.foo = 2;  
i = p.foo + 3;
```

There are three possible alias cases here:

1. The variables p and q cannot alias.
2. The variables p and q must alias.
3. It cannot be conclusively determined at compile time if p and q alias or not.

If p and q cannot alias, then `i = p.foo + 3;` can be changed to `i = 4`. If p and q must alias, then `i = p.foo + 3;` can be changed to `i = 5`. In both cases, we are able to perform optimizations from the alias knowledge. On the other hand, if it is not known if p and q alias or not, then no optimizations can be performed and the whole of the code must be executed to get the result. Two memory references are said to have a *may-alias* relation if their aliasing is unknown.

Performing Alias Analysis

In alias analysis, we divide the program's memory into *alias classes*. Alias classes are disjoint sets of locations that cannot alias to one another. For the discussion here, it is assumed that the optimizations done here occur on a low-level intermediate representation of the program. This is to say that the program has been compiled into binary operations, jumps, moves between registers, moves from registers to memory, moves from memory to registers, branches, and function calls/returns.

Type Based Alias Analysis

If the language being compiled is type safe, the compiler's type checker is correct, and the language lacks the ability to create pointers referencing local variables, (such as ML, Haskell, or Java) then some useful optimizations can be made. There are many cases where we know that two memory locations must be in different alias classes:

1. Two variables of different types cannot be in the same alias class since it is a property of strongly typed, memory reference-free (i.e. references to memory locations cannot be changed directly) languages that two variables of different types cannot share the same memory location simultaneously.
2. Allocations local to the current stack frame cannot be in the same alias class as any previous allocation from another stack frame. This is the case because new memory allocations must be disjoint from all other memory allocations.
3. Each record field of each record type has its own alias class, in general, because the typing discipline usually only allows for records of the same type to alias. Since all records of a type will be stored in an identical format in memory, a field can only alias to itself.
4. Similarly, each array of a given type has its own alias class.

When performing alias analysis for code, every load and store to memory needs to be labeled with its class. We then have the useful property, given memory locations A_i and B_j with i, j alias classes, that if $i = j$ then A_i may-alias B_j , and if $i \neq j$ then the memory locations will not alias.

Flow Based Alias Analysis

Analysis based on flow, unlike type based analysis, can be applied to programs in a language with references or type-casting. Flow based analysis can be used in lieu of or to supplement type based analysis. In flow based analysis, new alias classes are created for each memory allocation, and for every global and local variable whose address has been used. References may point to more than one value over time and thus may be in more than one alias class. This means that each memory location has a set of alias classes instead of a single alias class.

References

Appel, Andrew W. (1998). *Modern Compiler Implementation in ML*. Cambridge, UK: Cambridge University Press.
ISBN 0-521-60764-7.

External links

- Alias Analysis Library ^[1] - A simple C library for implementing alias analysis and a Master's Thesis giving an introduction to the field.

References

[1] <http://lenherr.name/~thomas/ma/>

Array access analysis

In computer science, **array access analysis** is a compiler analysis used to decide the read and write access patterns to elements or portions of arrays.

The major data type manipulated in scientific programs is the array. The define/use analysis on a whole array is insufficient for aggressive compiler optimizations such as auto parallelization and array privatization. Array access analysis aims to obtain the knowledge of which portions or even which elements of the array are accessed by a given code segment (basic block, loop, or even at the procedure level).

Array access analysis can be largely categorized into exact (or reference-list-based) and summary methods for different tradeoffs of accuracy and complexity. Exact methods are precise but very costly in terms of computation and space storage, while summary methods are approximate but can be computed quickly and economically.

Typical exact array access analysis include linearization and atom images. Summary methods can be further divided into array sections, bounded regular sections using triplet notation, linear-constraint methods such as data access descriptors and array region analysis.

References

Pointer analysis

In computer science **pointer analysis**, or **points-to analysis**, is a static code analysis technique that establishes which pointers, or heap references, can point to which variables or storage locations. It is often a component of more complex analyses such as escape analysis. A generalization of pointer analysis is shape analysis.

Introduction

Performing pointer analysis on all but the smallest of programs is a very resource intensive activity. Various simplifications have been made to reduce this overhead, the disadvantages of these simplifications is that the calculated set of objects pointed to may be larger than it is in practice.

Simplifications include:

Treating all references to a structured object as being to one object.

Ignoring flow-of control when analysing which objects are assigned to pointers, known as *context-insensitive pointer analysis* (when ignoring the context in which function calls are made) or *flow-insensitive pointer analysis* (when ignoring the control flow within a procedure).

Algorithms

- Steensgaard's algorithm
- Andersen's algorithm

References

- Michael Hind (2001). "Pointer analysis: haven't we solved this problem yet?" [1]. *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. pp. 54–61. ISBN 1-58113-413-4.

- Bjarne Steensgaard (1996). "Points-to analysis in almost linear time" [2]. *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. pp. 32–41. ISBN 0-89791-769-3.
- Andersen, Lars Ole (1994). *Program Analysis and Specialization for the C Programming Language* [3] (PhD thesis).

References

- [1] <http://www.cs.trinity.edu/~mlewis/CSCI3294-F01/Papers/p54-hind.pdf>
- [2] <http://cms.dc.uba.ar/materias/aap/2008/c2/descargas/pointsTo.pdf>
- [3] <http://eprints.kfupm.edu.sa/59416/>

Escape analysis

In programming language compiler optimization theory, **escape analysis** is a method for determining the dynamic scope of pointers. It is related to pointer analysis and shape analysis.

When a variable (or an object) is allocated in a subroutine, a pointer to the variable can *escape* to other threads of execution, or to calling subroutines. If a subroutine allocates an object and returns a pointer to it, the object can be accessed from undetermined places in the program — the pointer has "escaped". Pointers can also escape if they are stored in global variables or other data structures that, in turn, escape the current procedure.

Escape analysis determines all the places where a pointer can be stored and whether the lifetime of the pointer can be proven to be restricted only to the current procedure and/or thread.

Optimizations

A compiler can use the results of escape analysis as a basis for optimizations:

- *Converting heap allocations to stack allocations.* If an object is allocated in a subroutine, and a pointer to the object never escapes, the object may be a candidate for stack allocation instead of heap allocation.
- *Synchronization elision.* If an object is found to be accessible from one thread only, operations on the object can be performed without synchronization.
- *Breaking up objects or scalar replacement.* An object may be found to be accessed in ways that do not require the object to exist as a sequential memory structure. This may allow parts (or all) of the object to be stored in CPU registers instead of in memory.

Practical considerations

In object-oriented programming languages, dynamic compilers are particularly good candidates for performing escape analysis. In traditional static compilation, method overriding can make escape analysis impossible, as any called method might be overridden by a version that allows a pointer to escape. Dynamic compilers can perform escape analysis using the available information on overloading, and re-do the analysis when relevant methods are overridden by dynamic code loading.

The popularity of the Java programming language has made escape analysis a target of interest. Java's combination of heap-only object allocation, built-in threading, and the Sun HotSpot dynamic compiler creates a candidate platform for escape analysis related optimizations (see Escape analysis in Java). Escape analysis is implemented in Java Standard Edition 6.

Example (Java)

```
class A {
    final int finalValue;

    public A( B b ) {
        super();
        b.doSomething( this ); // this escapes!
        finalValue = 23;
    }

    int getTheValue() {
        return finalValue;
    }
}

class B {
    void doSomething( A a ) {
        System.out.println( a.getTheValue() );
    }
}
```

In this example, the constructor for class A passes the new instance of A to B.`doSomething`. As a result, the instance of A—and all of its fields—escapes the scope of the constructor.

Shape analysis

In program analysis, a **shape analysis** is a static code analysis technique that discovers and verifies properties of linked, dynamically allocated data structures in (usually imperative) computer programs. It is typically used at compile time to find software bugs or to verify high-level correctness properties of programs. In Java programs, it can be used to ensure that a sort method correctly sorts a list. For C programs, it might look for places where a block of memory is not properly freed. Although shape analyses are very powerful, they usually take a long time to run. For this reason, they have not seen widespread acceptance outside of universities and research labs (where they are only used experimentally).

Applications

Shape analysis has been applied to a variety of problems:

- Finding memory leaks, including Java-style leaks where a pointer to an unused object is not nulled out
- Discovering cases where a block of memory is freed more than once (in C)
- Finding dereferences of dangling pointers (pointers to freed memory in C)
- Finding array out-of-bounds errors
- Checking type-state properties (for example, ensuring that a file is `open()` before it is `read()`)
- Ensuring that a method to reverse a linked list does not introduce cycles into the list
- Verifying that a sort method returns a result that is in sorted order

Example

Shape analysis is a form of pointer analysis, although it is more precise than typical pointer analyses. Pointer analyses attempt to determine the set of objects to which a pointer can point (called the points-to set of the pointer). Unfortunately, these analyses are necessarily approximate (since a perfectly precise static analysis could solve the halting problem). Shape analyses can determine smaller (more precise) points-to sets.

Consider the following simple C++ program.

```
Item *items[10];
for (int i = 0; i < 10; ++i) {
    items[i] = new Item(...); // line [1]
}
process_items(items); // line [2]
for (int i = 0; i < 10; ++i) {
    delete items[i]; // line [3]
}
```

This program builds an array of objects, processes them in some arbitrary way, and then deletes them. Assuming that the `process_items` function is free of errors, it is clear that the program is safe: it never references freed memory, and it deletes all the objects that it has constructed.

Unfortunately, most pointer analyses have difficulty analyzing this program precisely. In order to determine points-to sets, a pointer analysis must be able to *name* a program's objects. In general, programs can allocate an unbounded number of objects; but in order to terminate, a pointer analysis can only use a finite set of names. A typical approximation is to give all the objects allocated on a given line of the program the same name. In the example above, all the objects constructed at line [1] would have the same name. Therefore, when the `delete` statement is analyzed for the first time, the analysis determines that one of the objects named [1] is being deleted. The second time the statement is analyzed (since it is in a loop) the analysis warns of a possible error: since it is unable to distinguish the objects in the array, it may be that the second `delete` is deleting the same object as the first `delete`. This warning is spurious, and the goal of shape analysis is to avoid such warnings.

Summarization and materialization

Shape analysis overcomes the problems of pointer analysis by using a more flexible naming system for objects. Rather than giving an object the same name throughout a program, objects can change names depending on the program's actions. Sometimes, several distinct objects with different names may be *summarized*, or merged, so that they have the same name. Then, when a summarized object is about to be used by the program, it can be *materialized*--that is, the summarized object is split into two objects with distinct names, one representing a single object and the other representing the remaining summarized objects. The basic heuristic of shape analysis is that objects that are being used by the program are represented using unique materialized objects, while objects not in use are summarized.

The array of objects in the example above is summarized in separate ways at lines [1], [2], and [3]. At line [1], the array has been only partly constructed. The array elements 0..i-1 contain constructed objects. The array element i is about to be constructed, and the following elements are uninitialized. A shape analysis can approximate this situation using a summary for the first set of elements, a materialized memory location for element i, and a summary for the remaining uninitialized locations, as follows:

0 .. i-1	i	i+1 .. 9
pointer to constructed object (summary)	uninitialized	uninitialized (summary)

After the loop terminates, at line [2], there is no need to keep anything materialized. The shape analysis determines at this point that all the array elements have been initialized:

0 .. 9
pointer to constructed object (summary)

At line [3], however, the array element `i` is in use again. Therefore, the analysis splits the array into three segments as in line [1]. This time, though, the first segment before `i` has been deleted, and the remaining elements are still valid (assuming the `delete` statement hasn't executed yet).

0 .. i-1	i	i+1 .. 9
free (summary)	pointer to constructed object	pointer to constructed object (summary)

Notice that in this case, the analysis recognizes that the pointer at index `i` has not been deleted yet. Therefore, it doesn't warn of a double deletion.

References

- Mooly Sagiv; Thomas Reps, Reinhard Wilhelm (May 2002). "Parametric shape analysis via 3-valued logic" ^[1]. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (ACM) **24** (3): 217–298.
doi:10.1145/292540.292552.

References

[1] <http://www.cs.wisc.edu/wpis/papers/toplas02.pdf>

Loop dependence analysis

In compiler theory, **loop dependence analysis** is the task of determining whether statements within a loop body form a dependence, with respect to array access and modification, induction, reduction and private variables, simplification of loop-independent code and management of conditional branches inside the loop body.

Loop dependence analysis is mostly done to find ways to do automatic parallelization, by means of vectorization, shared memory or others.

Description

Loop dependence analysis occur on a normalized loop of the form:

```
for i1 until U1 do
  for i2 until U2 do
    ...
    for in until Un do
      body
    done
  done
done
```

where `body` may contain:

```

S1  a[f1(i1, ..., in), ..., fm(i1, ..., in)] := ...
...
S2  ... := a[h1(i1, ..., in), ..., hm(i1, ..., in)]

```

Where a is an n -dimensional array and f_n , h_n , etc. are functions mapping from all iteration indexes (i_n) to a memory access in a particular dimension of the array.

For example, in C:

```

for (i = 0; i < U1; i++)
  for (j = 0; j < U2; j++)
    a[i+4-j] = b[2*i-j] + i*j;

```

f_1 would be $i+4-j$, controlling the write on the first dimension of a and f_2 would be $2*i-j$, controlling the read on the first dimension of b .

The scope of the problem is to find all possible dependencies between $S1$ and $S2$. To be conservative, any dependence which cannot be proven false must be assumed to be true.

Independence is shown by demonstrating that no two instances of $S1$ and $S2$ access or modify the same spot in array a . When a possible dependence is found, loop dependence analysis usually makes every attempt to characterize the relationship between dependent instances, as some optimizations may still be possible. It may also be possible to transform the loop to remove or modify the dependence.

In the course of (dis)proving such dependencies, a statement S may be decomposed according to which iteration it comes from. For instance, $S[1,3,5]$ refers to the iteration where $i_1 = 1$, $i_2 = 3$ and $i_3 = 5$. Of course, references to abstract iterations, such as $S[d1+1,d2,d3]$, are both permitted and common.

Iteration vectors

A specific iteration through a normalized loop is referenced through an **iteration vector**, which encodes the state of each iteration variable.

For a loop, an iteration vector is a member of the Cartesian product of the bounds for the loop variables. In the normalized form given previously, this space is defined to be $U1 \times U2 \times \dots \times Un$. Specific instances of statements may be parametrized by these iteration vectors, and they are also the domain of the array subscript functions found in the body of the loop. Of particular relevance, these vectors form a lexicographic order which corresponds with the chronological execution order.

Dependence Vectors

To classify data dependence, compilers use two important vectors: the **distance vector** (σ), which indicates the distance between f_n and h_n , and the **direction vector** (ρ), which indicates the corresponding direction, basically the sign of the distance.

The **distance vector** is defined as $\sigma = (\sigma_1, \dots, \sigma_k)$ where σ_n is $\sigma_n = i_n - j_n$

The **direction vector** is defined as $\rho = (\rho_1, \dots, \rho_k)$ where ρ_n is:

- ($<$) if $\sigma_n > 0 \Rightarrow [f_n < h_n]$
- ($=$) if $\sigma_n = 0 \Rightarrow [f_n = h_n]$
- ($>$) if $\sigma_n < 0 \Rightarrow [f_n > h_n]$

Classification

A dependence between two operations: a and b , can be classified according to the following criteria:

- **Operation type**
 - If a is a write and b is a read, this is a flow dependence
 - If a is a read and b is a write, this is an anti-dependence
 - If a is a write and b is a write, this is an output dependence
 - If a is a read and b is a read, this is an input dependence
- **Chronological order**
 - If $S_a < S_b$, this is a lexically forward dependence
 - If $S_a = S_b$, this is a self-dependence
 - If $S_a > S_b$, this is a lexically backward dependence
- **Loop dependence**
 - If all distances (σ) are zero (same place in memory), this is loop independent
 - If at least one distance is non-zero, this is a loop carried dependence

Plausibility

Some loop dependence relations can be parallelized (or vectorized) and some cannot. Each case must be analysed separately, but as a general rule of thumb, the following table cover most cases:

$\rho \setminus \text{order}$	Lexically forward	Self-dependence	Lexically backward
positive (<)	plausible	plausible	plausible
zero (=)	implausible	δ^a : plausible δ^f : implausible	plausible
negative (>)	implausible	implausible	implausible

Some implausible dependences can be transformed into plausible ones, for example, by means of re-arranging the statements.

Alias detection

Inside loops, the same variable can be accessed for both read and write, at the same or different location within each iteration. Not only that, but the same region in memory can be accessed via different variables. When the same region in memory can be accessed by more than one variable, you have an alias.

Some aliases are very simple to detect:

```
a = b;
for (i = 0; i < MAX; ++i)
    a[i] = b[i+1];
a[MAX] = 0;
```

It is obvious that **b** is an alias to **a**, thus this code is actually *shifting* the array to the left. But this is not always so obvious, for instance the standard C library function *strcpy()* copies one string to another, but the caller could provide overlapped regions like this:

```
strcpy(x, x+1);
```

when the internal loop could be implemented as:

```
while (*src != 0) {
    *dst = *src;
    src++; dst++;
}
```

The dependency of **src** and **dst** is not obvious from within the function, you have to analyse every caller to make sure there isn't any. In the case of a library function, there is no way to assure it won't happen, so the compiler has to assume one way or another. If the compiler assumes there is no alias, whenever the regions overlap, you get undefined behaviour. If it assumes there is, you always get non-optimized code for every case.

Some compilers accept special keywords to work out if it can assume no alias, such as *restrict*.

Techniques

Several established devices and techniques exist for tackling the loop dependence problem. For determining whether a dependence exists, the GCD test and the Banerjee test are the most general tests in common use, while a variety of techniques exist for simpler cases.

Further reading

- Kennedy, Ken; & Allen, Randy. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann. ISBN 1-55860-286-0.
- Muchnick, Steven S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann. ISBN 1-55860-320-4.
- Bik, Aart J.C. (2004). *The Software Vectorization Handbook*. Intel press. ISBN 0-9743649-2-4.

Program slicing

In computer programming, **program slicing** is the computation of the set of programs statements, the **program slice**, that may affect the values at some point of interest, referred to as a **slicing criterion**. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control.

Slicing techniques have been seeing a rapid development since the original definition by Mark Weiser. At first, slicing was only static, i.e., applied on the source code with no other information than the source code. Bogdan Korel and Janusz Laski introduced *dynamic slicing*, which works on a specific execution of the program (for a given execution trace). Other forms of slicing exist, for instance path slicing.

Static slicing

Based on the original definition of Weiser, informally, a static program slice S consists of all statements in program P that may affect the value of variable v at some point p . The slice is defined for a slicing criterion $C=(x,V)$, where x is a statement in program P and V is a subset of variables in P . A static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement x). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

Example

```
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i) {
    sum = sum + i;
    product = product *i;
}
write(sum);
write(product);
```

This new program is a valid slicing of the above program with respect to the criterion (`write(sum),{sum}`):

```
int i;
int sum = 0;

for(i = 0; i < N; ++i) {
    sum = sum + i;

}

write(sum);
```

In fact, most static slicing techniques, including Weiser's own technique, will also remove the `write(sum)` statement. Indeed, at the statement `write(sum)`, the value of `sum` is not dependent on the statement itself.

Dynamic slicing

Makes use of information about a particular execution of a program. A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program.

An example to clarify the difference between static and dynamic slicing. Consider a small piece of a program unit, in which there is an iteration block containing an if-else block. There are a few statements in both the `if` and `else` blocks that have an effect on a variable. In the case of static slicing, since the whole program unit is looked at irrespective of a particular execution of the program, the affected statements in both blocks would be included in the slice. But, in the case of dynamic slicing we consider a particular execution of the program, wherein the `if` block gets executed and the affected statements in the `else` block do not get executed. So, in this particular execution case, the dynamic slice would contain only the statements in the `if` block.

References

- Mark Weiser. "Program slicing". Proceedings of the 5th International Conference on Software Engineering, pages 439–449, IEEE Computer Society Press, March 1981.
- Mark Weiser. "Program slicing". IEEE Transactions on Software Engineering, Volume 10, Issue 4, pages 352–357, IEEE Computer Society Press, July 1984.
- Frank Tip. "A survey of program slicing techniques". Journal of Programming Languages, Volume 3, Issue 3, pages 121–189, September 1995.
- David Binkley and Keith Brian Gallagher. "Program slicing". Advances in Computers, Volume 43, pages 1–50, Academic Press, 1996.
- Andrea de Lucia. "Program slicing: Methods and applications", International Workshop on Source Code Analysis and Manipulation, pages 142-149, 2001, IEEE Computer Society Press.
- Mark Harman and Robert Hierons. "An overview of program slicing", Software Focus, Volume 2, Issue 3, pages 85–92, January 2001.
- David Binkley and Mark Harman. "A survey of empirical results on program slicing", Advances in Computers, Volume 62, pages 105-178, Academic Press, 2004.
- Jens Krinke. "Program Slicing", In Handbook of Software Engineering and Knowledge Engineering, Volume 3: Recent Advances. World Scientific Publishing, 2005

External links

- VALSOFT/Joana Project ^[1]
- Indus Project ^[2] (part of Bandera checker)
- Wisconsin Program-Slicing Project ^[3]
- SoftwareMining Extracting Business Rules/Program Slices from COBOL programs. (Web: www.softwaremining.com ^[4])
- StaticSlicer, a simple tool which demonstrates static slicing based on the original definition of Weiser, ^[5]

References

[1] <http://pp.info.uni-karlsruhe.de/project.php?id=30>

[2] <http://indus.projects.cis.ksu.edu/index.shtml>

[3] <http://www.cs.wisc.edu/wpis/html/>

[4] <http://www.softwaremining.com/>

[5] <http://sourceforge.net/projects/someslice/>

Code optimization

Compiler optimization

Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of *optimizing transformations*, algorithms which take a program and transform it to produce a semantically equivalent output program that uses less resources.

It has been shown that some code optimization problems are NP-complete, or even undecidable. In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implementor might provide. (Optimization is generally a very CPU- and memory-intensive process.) In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed. Because of all these factors, optimization rarely produces "optimal" output in any sense, and in fact an "optimization" may impede performance in some cases; rather, they are heuristic methods for improving resource usage in typical programs.

Types of optimizations

Techniques used in optimization can be broken up among various *scopes* which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

Peephole optimizations

Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself. (This example is also an instance of strength reduction.)

Local optimizations

These only consider information local to a function definition. This reduces the amount of analysis that needs to be performed (saving time and reducing storage requirements) but means that worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

Loop optimizations

These act on the statements which make up a loop, such as a *for* loop (e.g., loop-invariant code motion). Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

Interprocedural or whole-program optimization

These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information (i.e., within a single function). This kind of optimization can also allow new techniques to be performed. For instance function inlining, where a call to a function is replaced by a copy of the function body.

Machine code optimization

These analyze the executable task image of the program after all of a executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression (which saves space by collapsing common sequences of instructions), are more effective when the entire executable task image is available for analysis.^[1]

In addition to scoped optimizations there are two further general categories of optimization:

Programming language-independent vs language-dependent

Most high-level languages share common programming constructs and abstractions: decision (if, switch, case), looping (for, while, repeat.. until, do.. while), and encapsulation (structures, objects). Thus similar optimization techniques can be used across languages. However, certain language features make some kinds of optimizations difficult. For instance, the existence of pointers in C and C++ makes it difficult to optimize array accesses (see Alias analysis). However, languages such as PL/1 (that also supports pointers) nevertheless have available sophisticated optimizing compilers to achieve better performance in various other ways. Conversely, some language features make certain optimizations easier. For example, in some languages functions are not permitted to have side effects. Therefore, if a program makes several calls to the same function with the same arguments, the compiler can immediately infer that the function's result need be computed only once.

Machine independent vs machine dependent

Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler, but many of the most effective optimizations are those that best exploit special features of the target platform.

The following is an instance of a local machine dependent optimization. To set a register to 0, the obvious way is to use the constant '0' in an instruction that sets a register value to a constant. A less obvious way is to XOR a register with itself. It is up to the compiler to know which instruction variant to use. On many RISC machines, both instructions would be equally appropriate, since they would both be the same length and take the same time. On many other microprocessors such as the Intel x86 family, it turns out that the XOR variant is shorter and probably faster, as there will be no need to decode an immediate operand, nor use the internal "immediate operand register". (A potential problem with this is that XOR may introduce a data dependency on the previous value of the register, causing a pipeline stall. However, processors often have XOR of a register with itself as a special case that doesn't cause stalls.)

Factors affecting optimization

The machine itself

Many of the choices about which optimizations can and should be done depend on the characteristics of the target machine. It is sometimes possible to parameterize some of these machine dependent factors, so that a single piece of compiler code can be used to optimize different machines just by altering the machine description parameters. GCC is a compiler which exemplifies this approach.

The architecture of the target CPU

Number of CPU registers: To a certain extent, the more registers, the easier it is to optimize for performance. Local variables can be allocated in the registers and not on the stack. Temporary/intermediate results can be left in registers without writing to and reading back from memory.

- RISC vs CISC: CISC instruction sets often have variable instruction lengths, often have a larger number of possible instructions that can be used, and each instruction could take differing amounts of time. RISC instruction sets attempt to limit the variability in each of these: instruction sets are usually constant length, with few exceptions, there are usually fewer combinations of registers and memory operations, and the instruction issue rate (the number of instructions completed per time period, usually an integer multiple of the clock cycle)

is usually constant in cases where memory latency is not a factor. There may be several ways of carrying out a certain task, with CISC usually offering more alternatives than RISC. Compilers have to know the relative costs among the various instructions and choose the best instruction sequence (see instruction selection).

- Pipelines: A pipeline is essentially a CPU broken up into an assembly line. It allows use of parts of the CPU for different instructions by breaking up the execution of instructions into various stages: instruction decode, address decode, memory fetch, register fetch, compute, register store, etc. One instruction could be in the register store stage, while another could be in the register fetch stage. Pipeline conflicts occur when an instruction in one stage of the pipeline depends on the result of another instruction ahead of it in the pipeline but not yet completed. Pipeline conflicts can lead to pipeline stalls: where the CPU wastes cycles waiting for a conflict to resolve.

Compilers can *schedule*, or reorder, instructions so that pipeline stalls occur less frequently.

- Number of functional units: Some CPUs have several ALUs and FPUs. This allows them to execute multiple instructions simultaneously. There may be restrictions on which instructions can pair with which other instructions ("pairing" is the simultaneous execution of two or more instructions), and which functional unit can execute which instruction. They also have issues similar to pipeline conflicts.

Here again, instructions have to be scheduled so that the various functional units are fully fed with instructions to execute.

The architecture of the machine

- Cache size (256 kiB–12 MiB) and type (direct mapped, 2-/4-/8-/16-way associative, fully associative): Techniques such as inline expansion and loop unrolling may increase the size of the generated code and reduce code locality. The program may slow down drastically if a highly utilized section of code (like inner loops in various algorithms) suddenly cannot fit in the cache. Also, caches which are not fully associative have higher chances of cache collisions even in an unfilled cache.
- Cache/Memory transfer rates: These give the compiler an indication of the penalty for cache misses. This is used mainly in specialized applications.

Intended use of the generated code

Debugging

While a programmer is writing an application, he will recompile and test often, and so compilation must be fast. This is one reason most optimizations are deliberately avoided during the test/debugging phase. Also, program code is usually "stepped through" (see Program animation) using a symbolic debugger, and optimizing transformations, particularly those that reorder code, can make it difficult to relate the output code with the line numbers in the original source code. This can confuse both the debugging tools and the programmers using them.

General purpose use

Prepackaged software is very often expected to be executed on a variety of machines and CPUs that may share the same instruction set, but have different timing, cache or memory characteristics. So, the code may not be tuned to any particular CPU, or may be tuned to work best on the most popular CPU and yet still work acceptably well on other CPUs.

Special-purpose use

If the software is compiled to be used on one or a few very similar machines, with known characteristics, then the compiler can heavily tune the generated code to those specific machines (if such options are available). Important special cases include code designed for parallel and vector processors, for which special parallelizing compilers are employed.

Embedded systems

These are a common case of special-purpose use. Embedded software can be tightly tuned to an exact CPU and memory size. Also, system cost or reliability may be more important than the code's speed. So, for example, compilers for embedded software usually offer options that reduce code size at the expense of speed, because memory is the main cost of an embedded computer. The code's timing may need to be predictable, rather than as fast as possible, so code caching might be disabled, along with compiler optimizations that require it.

Common themes

To a large extent, compiler optimization techniques have the following themes, which sometimes conflict.

Optimize the common case

The common case may have unique properties that allow a *fast path* at the expense of a *slow path*. If the fast path is taken most often, the result is better over-all performance.

Avoid redundancy

Reuse results that are already computed and store them for use later, instead of recomputing them.

Less code

Remove unnecessary computations and intermediate values. Less work for the CPU, cache, and memory usually results in faster execution. Alternatively, in embedded systems, less code brings a lower product cost.

Fewer jumps by using *straight line code*, also called *branch-free code*

Less complicated code. Jumps (conditional or unconditional branches) interfere with the prefetching of instructions, thus slowing down code. Using inlining or loop unrolling can reduce branching, at the cost of increasing binary file size by the length of the repeated code. This tends to merge several basic blocks into one.

Locality

Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference.

Exploit the memory hierarchy

Accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk.

Parallelize

Reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.

More precise information is better

The more precise the information the compiler has, the better it can employ any or all of these optimization techniques.

Runtime metrics can help

Information gathered during a test run can be used in profile-guided optimization. Information gathered at runtime (ideally with minimal overhead) can be used by a JIT compiler to dynamically improve optimization.

Strength reduction

Replace complex or difficult or expensive operations with simpler ones. For example, replacing division by a constant with multiplication by its reciprocal, or using induction variable analysis to replace multiplication by a loop index with addition.

Specific techniques

Loop optimizations

Some optimization techniques primarily designed to operate on loops include:

Induction variable analysis

Roughly, if a variable in a loop is a simple function of the index variable, such as $j := 4*i + 1$, it can be updated appropriately each time the loop variable is changed. This is a strength reduction, and also may allow the index variable's definitions to become dead code. This information is also useful for bounds-checking elimination and dependence analysis, among other things.

Loop fission or loop distribution

Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.

Loop fusion or loop combining

Another technique which attempts to reduce loop overhead. When two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data.

Loop inversion

This technique changes a standard *while* loop into a *do/while* (also known as *repeat/until*) loop wrapped in an *if* conditional, reducing the number of jumps by two, for cases when the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the *if* guard can be skipped.

Loop interchange

These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.

Loop-invariant code motion

If a quantity is computed inside a loop during every iteration, and its value is the same for each iteration, it can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins. This is particularly important with the address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with loop inversion, because not all code is safe to be hoisted outside the loop.

Loop nest optimization

Some pervasive algorithms such as matrix multiplication have very poor cache behavior and excessive memory accesses. Loop nest optimization increases the number of cache hits by performing the operation over small blocks and by using a loop interchange.

Loop reversal

Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations.

Loop unrolling

Unrolling duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which hurt performance by impairing the instruction pipeline. A "fewer jumps" optimization. Completely unrolling a loop eliminates all overhead, but requires that the number of iterations be known at compile time.

Loop splitting

Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. A useful special case is *loop peeling*, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

Loop unswitching

Unswitching moves a conditional from inside a loop to outside the loop by duplicating the loop's body inside each of the if and else clauses of the conditional.

Software pipelining

The loop is restructured in such a way that work done in an iteration is split into several parts and done over several iterations. In a tight loop this technique hides the latency between loading and using values.

Automatic parallelization

A loop is converted into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine, including multi-core machines.

Data-flow optimizations

Data flow optimizations, based on Data-flow analysis, primarily depend on how certain properties of data are propagated by control edges in the control flow graph. Some of these include:

Common subexpression elimination

In the expression "(a + b) - (a + b)/4", "common subexpression" refers to the duplicated "(a + b)". Compilers implementing this technique realize that "(a+b)" won't change, and as such, only calculate its value once.

Constant folding and propagation

replacing expressions consisting of constants (e.g., "3 + 5") with their final value ("8") at compile time, rather than doing the calculation in run-time. Used in most modern languages.

Induction variable recognition and elimination

see discussion above about *induction variable analysis*.

Alias classification and pointer analysis

in the presence of pointers, it is difficult to make any optimizations at all, since potentially any variable can have been changed when a memory location is assigned to. By specifying which pointers can alias which variables, unrelated pointers can be ignored.

Dead store elimination

removal of assignments to variables that are not subsequently read, either because the lifetime of the variable ends or because of a subsequent assignment that will overwrite the first value.

SSA-based optimizations

These optimizations are intended to be done after transforming the program into a special form called static single assignment (see SSA form), in which every variable is assigned in only one place. Although some function without SSA, they are most effective with SSA. Many optimizations listed in other sections also benefit with no special changes, such as register allocation.

Global value numbering

GVN eliminates redundancy by constructing a value graph of the program, and then determining which values are computed by equivalent expressions. GVN is able to identify some redundancy that common subexpression elimination cannot, and vice versa.

Sparse conditional constant propagation

Effectively equivalent to iteratively performing constant propagation, constant folding, and dead code elimination until there is no change, but is much more efficient. This optimization symbolically executes the program, simultaneously propagating constant values and eliminating portions of the control flow graph that this makes unreachable.

Code generator optimizations

Register allocation

The most frequently used variables should be kept in processor registers for fastest access. To find which variables to put in registers an interference-graph is created. Each variable is a vertex and when two variables are used at the same time (have an intersecting live range) they have an edge between them. This graph is colored using for example Chaitin's algorithm using the same number of colors as there are registers. If the coloring fails one variable is "spilled" to memory and the coloring is retried.

Instruction selection

Most architectures, particularly CISC architectures and those with many addressing modes, offer several different ways of performing a particular operation, using entirely different sequences of instructions. The job of the instruction selector is to do a good job overall of choosing which instructions to implement which operators in the low-level intermediate representation with. For example, on many processors in the 68000 family and on the x86 architecture, complex addressing modes can be used in statements like "lea 25(a1,d5*4), a0", allowing a single instruction to perform a significant amount of arithmetic with less storage.

Instruction scheduling

Instruction scheduling is an important optimization for modern pipelined processors, which avoids stalls or bubbles in the pipeline by clustering instructions with no dependencies together, while being careful to preserve the original semantics.

Rematerialization

Rematerialization recalculates a value instead of loading it from memory, preventing a memory access. This is performed in tandem with register allocation to avoid spills.

Code factoring

If several sequences of code are identical, or can be parameterized or reordered to be identical, they can be replaced with calls to a shared subroutine. This can often share code for subroutine set-up and sometimes tail-recursion.^[2]

Trampolines

Many CPUs have smaller subroutine call instructions to access low memory. A compiler can save space by using these small calls in the main body of code. Jump instructions in low memory can access the routines at any address. This multiplies space savings from code factoring.^[2]

Reordering computations

Based on integer linear programming, restructuring compilers enhance data locality and expose more parallelism by reordering computations. Space-optimizing compilers may reorder code to lengthen sequences that can be factored into subroutines.

Functional language optimizations

Although many of these also apply to non-functional languages, they either originate in, are most easily implemented in, or are particularly critical in functional languages such as Lisp and ML.

Removing recursion

Recursion is often expensive, as a function call consumes stack space and involves some overhead related to parameter passing and flushing the instruction cache. Tail recursive algorithms can be converted to iteration, which does not have call overhead and uses a constant amount of stack space, through a process called tail recursion elimination or tail call optimization. Some functional languages (e.g., Scheme and Erlang) mandate that tail calls be optimized by a conforming implementation, due to their prevalence in these languages.

Deforestation (data structure fusion)

Because of the high level nature by which data structures are specified in functional languages such as Haskell, it is possible to combine several recursive functions which produce and consume some temporary data structure so that the data is passed directly without wasting time constructing the data structure.

Other optimizations

Please help separate and categorize these further and create detailed pages for them, especially the more complex ones, or link to one where one exists.

Bounds-checking elimination

Many languages, for example Java, enforce bounds-checking of all array accesses. This is a severe performance bottleneck on certain applications such as scientific code. Bounds-checking elimination allows the compiler to safely remove bounds-checking in many situations where it can determine that the index must fall within valid bounds, for example if it is a simple loop variable.

Branch offset optimization (machine independent)

Choose the shortest branch displacement that reaches target

Code-block reordering

Code-block reordering alters the order of the basic blocks in a program in order to reduce conditional branches and improve locality of reference.

Dead code elimination

Removes instructions that will not affect the behaviour of the program, for example definitions which have no uses, called dead code. This reduces code size and eliminates unnecessary computation.

Factoring out of invariants

If an expression is carried out both when a condition is met and is not met, it can be written just once outside of the conditional statement. Similarly, if certain types of expressions (e.g., the assignment of a constant into a variable) appear inside a loop, they can be moved out of it because their effect will be the same no matter if they're executed many times or just once. Also known as total redundancy elimination. A more powerful optimization is partial redundancy elimination (PRE).

Inline expansion or macro expansion

When some code invokes a procedure, it is possible to directly insert the body of the procedure inside the calling code rather than transferring control to it. This saves the overhead related to procedure calls, as well as

providing great opportunity for many different parameter-specific optimizations, but comes at the cost of space; the procedure body is duplicated each time the procedure is called inline. Generally, inlining is useful in performance-critical code that makes a large number of calls to small procedures. A "fewer jumps" optimization.

Jump threading

In this pass, consecutive conditional jumps predicated entirely or partially on the same condition are merged. E.g., `<syntaxhighlight lang="text" enclose="none"> if (c) { foo; } if (c) { bar; } </syntaxhighlight> to <syntaxhighlight lang="text" enclose="none"> if (c) { foo; bar; } </syntaxhighlight>, and <syntaxhighlight lang="text" enclose="none"> if (c) { foo; } if (!c) { bar; } </syntaxhighlight> to <syntaxhighlight lang="text" enclose="none"> if (c) { foo; } else { bar; } </syntaxhighlight>.`

Macro Compression

A space optimization that recognizes common sequences of code, creates subprograms ("code macros") that contain the common code, and replaces the occurrences of the common code sequences with calls to the corresponding subprograms.^[1] This is most effectively done as a machine code optimization, when all the code is present. The technique was first used to conserve space in an interpretive byte stream used in an implementation of Macro Spitbol on microcomputers.^[3] The problem of determining an optimal set of macros that minimizes the space required by a given code segment is known to be NP-Complete,^[1] but efficient heuristics attain near-optimal results.^[4]

Reduction of cache collisions

(e.g., by disrupting alignment within a page)

Stack height reduction

Rearrange expression tree to minimize resources needed for expression evaluation.

Test reordering

If we have two tests that are the condition for something, we can first deal with the simpler tests (e.g. comparing a variable to something) and only then with the complex tests (e.g., those that require a function call). This technique complements lazy evaluation, but can be used only when the tests are not dependent on one another. Short-circuiting semantics can make this difficult.

Interprocedural optimizations

Interprocedural optimization works on the entire program, across procedure and file boundaries. It works tightly with intraprocedural counterparts, carried out with the cooperation of a local part and global part. Typical interprocedural optimizations are: procedure inlining, interprocedural dead code elimination, interprocedural constant propagation, and procedure reordering. As usual, the compiler needs to perform interprocedural analysis before its actual optimizations. Interprocedural analyses include alias analysis, array access analysis, and the construction of a call graph.

Interprocedural optimization is common in modern commercial compilers from SGI, Intel, Microsoft, and Sun Microsystems. For a long time the open source GCC was criticized for a lack of powerful interprocedural analysis and optimizations, though this is now improving. Another good open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and for commercial purposes.

Due to the extra time and space required by interprocedural analysis, most compilers do not perform it by default. Users must use compiler options explicitly to tell the compiler to enable interprocedural analysis and other expensive optimizations.

Problems with optimization

Early in the history of compilers, compiler optimizations were not as good as hand-written ones. As compiler technologies have improved, good compilers can often generate better code than human programmers, and good post pass optimizers can improve highly hand-optimized code even further. For RISC CPU architectures, and even more so for VLIW hardware, compiler optimization is the key for obtaining efficient code, because RISC instruction sets are so compact that it is hard for a human to manually schedule or combine small instructions to get efficient results. Indeed, these architectures were designed to rely on compiler writers for adequate performance.

However, optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all program source code, the fastest (or smallest) possible equivalent compiled program is output; such a compiler is fundamentally impossible because it would solve the halting problem.

This may be proven by considering a call to a function, `foo()`. This function returns nothing and does not have side effects (no I/O, does not modify global variables and "live" data structures, etc.). The fastest possible equivalent program would be simply to eliminate the function call. However, if the function `foo()` in fact does *not* return, then the program with the call to `foo()` would be different from the program without the call; the optimizing compiler will then have to determine this by solving the halting problem.

Additionally, there are a number of other more practical issues with optimizing compiler technology:

- Optimizing compilers focus on relatively shallow constant-factor performance improvements and do not typically improve the algorithmic complexity of a solution. For example, a compiler will not change an implementation of bubble sort to use mergesort instead.
- Compilers usually have to support a variety of conflicting objectives, such as cost of implementation, compilation speed and quality of generated code.
- A compiler typically only deals with a part of a program at a time, often the code contained within a single file or module; the result is that it is unable to consider contextual information that can only be obtained by processing the other files.
- The overhead of compiler optimization: Any extra work takes time; whole-program optimization is time consuming for large programs.
- The often complex interaction between optimization phases makes it difficult to find an optimal sequence in which to execute the different optimization phases.

Work to improve optimization technology continues. One approach is the use of so-called post-pass optimizers (some commercial versions of which date back to mainframe software of the late 1970s^[5]). These tools take the executable output by an "optimizing" compiler and optimize it even further. Post pass optimizers usually work on the assembly language or machine code level (contrast with compilers that optimize intermediate representations of programs). The performance of post pass compilers are limited by the fact that much of the information available in the original source code is not always available to them.

As processor performance continues to improve at a rapid pace, while memory bandwidth improves more slowly, optimizations that reduce memory bandwidth (even at the cost of making the processor execute relatively more instructions) will become more useful. Examples of this, already mentioned above, include loop nest optimization and rematerialization.

List of compiler optimizations

- Automatic programming
- automatic parallelization
- Constant folding
- Algebraic simplifications:
- Value numbering
- Copy propagation
- Constant propagation
- Sparse conditional constant propagation
- Common subexpression elimination (CSE)
- Partial redundancy elimination
- Dead code elimination
- Induction variable elimination, strength reduction
- Loop optimizations
 - Loop invariant code motion
 - Loop unrolling
- Software pipelining
- Inlining
- Code generator
 - Register allocation: local and global
 - Instruction scheduling
 - Branch predication
 - Tail merging and cross jumping
 - Machine idioms and instruction combining
- Vectorization
- Phase ordering
- Profile-guided optimization
- Macro compression

List of static code analyses

- Alias analysis
- Pointer analysis
- Shape analysis
- Escape analysis
- Array access analysis
- Dependence analysis
- Control flow analysis
- Data flow analysis
 - Use-define chain analysis
 - Live variable analysis
 - Available expression analysis

References

- [1] Clinton F. Goss (June 1986). "Machine Code Optimization - Improving Executable Object Code" (<http://www.ClintGoss.com/mco/>). pp. 112. . Retrieved 24-Mar-2011.
- [2] Cx51 Compiler Manual, version 09.2001, p155, Keil Software Inc.
- [3] Robert B. K. Dewar; Martin Charles Golumbic; Clinton F. Goss (October 1979). *MICRO SPITBOL*. Computer Science Department Technical Report. No. 11. Courant Institute of Mathematical Sciences.
- [4] Martin Charles Golumbic; Robert B. K. Dewar; Clinton F. Goss (1980). "Macro Substitutions in MICRO SPITBOL - a Combinatorial Analysis". *Proc. 11th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium, Utilitas Math., Winnipeg, Canada* 29: 485–495.
- [5] <http://portal.acm.org/citation.cfm?id=358728.358732>

External links

- Optimization manuals (<http://www.agner.org/optimize/#manuals>) by Agner Fog - documentation about x86 processor architecture and low-level code optimization
- Citations from CiteSeer (<http://citeseer.ist.psu.edu/Programming/CompilerOptimization/>)

Peephole optimization

In compiler theory, **peephole optimization** is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognising sets of instructions that don't actually do anything, or that can be replaced by a leaner set of instructions.

Replacement rules

Common techniques applied in peephole optimization:^[1]

- Constant folding - Evaluate constant subexpressions in advance.
- Strength reduction - Replace slow operations with faster equivalents.
- Null sequences - Delete useless operations
- Combine Operations: Replace several operations with one equivalent.
- Algebraic Laws: Use algebraic laws to simplify or reorder instructions.
- Special Case Instructions: Use instructions designed for special operand cases.
- Address Mode Operations: Use address modes to simplify code.

There can, of course, be other types of peephole optimizations involving simplifying the target machine instructions, assuming that the target machine is known in advance. Advantages of a given architecture and instruction sets can be exploited in this case.

Examples

Replacing slow instructions with faster ones

The following Java bytecode

```
...
aload 1
aload 1
mul
...
```

can be replaced by

```
...
aload 1
dup
mul
...
```

This kind of optimization, like most peephole optimizations, makes certain assumptions about the efficiency of instructions. For instance, in this case, it is assumed that the `dup` operation (which duplicates and pushes the top of the stack) is more efficient than the `aload X` operation (which loads a local variable identified as `X` and pushes it on the stack).

Removing redundant code

Another example is to eliminate redundant load stores.

```
a = b + c;
d = a + e;
```

is straightforwardly implemented as

```
MOV b, R0 # Copy b to the register
ADD c, R0 # Add c to the register, the register is now b+c
MOV R0, a # Copy the register to a
MOV a, R0 # Copy a to the register
ADD e, R0 # Add e to the register, the register is now a+e [ (b+c)+e]
MOV R0, d # Copy the register to d
```

but can be optimised to

```
MOV b, R0 # Copy b to the register
ADD c, R0 # Add c to the register, which is now b+c (a)
MOV R0, a # Copy the register to a
ADD e, R0 # Add e to the register, which is now b+c+e [ (a)+e]
MOV R0, d # Copy the register to d
```

Furthermore, if the compiler knew that the variable `a` was not used again, the middle operation could be omitted.

Removing redundant stack instructions

If the compiler saves registers on the stack before calling a subroutine and restores them when returning, consecutive calls to subroutines may have redundant stack instructions.

Suppose the compiler generates the following Z80 instructions for each procedure call:

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL _ADDR
POP HL
POP DE
POP BC
POP AF
```

If there were two consecutive subroutine calls, they would look like this:

```
PUSH AF  
PUSH BC  
PUSH DE  
PUSH HL  
CALL _ADDR1  
POP HL  
POP DE  
POP BC  
POP AF  
PUSH AF  
PUSH BC  
PUSH DE  
PUSH HL  
CALL _ADDR2  
POP HL  
POP DE  
POP BC  
POP AF
```

The sequence POP reg followed by PUSH for the same registers is generally redundant. In cases where it is redundant, a peephole optimization would remove these instructions. In the example, this would cause another redundant POP/PUSH pair to appear in the peephole, and these would be removed in turn. Removing all of the redundant code in the example above would eventually leave the following code:

```
PUSH AF  
PUSH BC  
PUSH DE  
PUSH HL  
CALL _ADDR1  
CALL _ADDR2  
POP HL  
POP DE  
POP BC  
POP AF
```

Implementation

Modern architectures typically allow for many hundreds of different kinds of peephole optimizations, and it is therefore often appropriate for compiler programmers to implement them using a pattern matching algorithm.^[2]

References

- [1] Crafting a Compiler with C++, Fischer/LeBlanc
- [2] Compilers - Principles, Techniques, and Tools 2e, p560

External links

- The copt general-purpose peephole optimizer by Christopher W. Fraser (<ftp://ftp.cs.princeton.edu/pub/lcc/contrib/copt.shar>)
- The original paper (<http://portal.acm.org/citation.cfm?id=365000>)

Copy propagation

In compiler theory, **copy propagation** is the process of replacing the occurrences of targets of direct assignments with their values^[1]. A direct assignment is an instruction of the form $x = y$, which simply assigns the value of y to x .

From the following code:

```
y = x  
z = 3 + y
```

Copy propagation would yield:

```
z = 3 + x
```

Copy propagation often makes use of reaching definitions, use-def chains and def-use chains when computing which occurrences of the target may be safely replaced. If all upwards exposed uses of the target may be safely modified, the assignment operation may be eliminated.

Copy propagation is a useful "clean up" optimization frequently used after other optimizations have already been run. Some optimizations -- such as elimination of common sub expressions^[1] -- require that copy propagation be run afterwards in order to achieve an increase in efficiency.

References

- [1] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D.. *Compilers, Principles, Techniques, & Tools Second edition*. ISBN 0-321-48681-1.

Further reading

Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann. 1997.

Constant folding

Constant folding and **constant propagation** are related compiler optimizations used by many modern compilers. An advanced form of constant propagation known as sparse conditional constant propagation can more accurately propagate constants and simultaneously remove dead code.

Constant folding

Constant folding is the process of simplifying constant expressions at compile time. Terms in constant expressions are typically simple literals, such as the integer 2, but can also be variables whose values are never modified, or variables explicitly marked as constant. Consider the statement:

```
i = 320 * 200 * 32;
```

Most modern compilers would not actually generate two multiply instructions and a store for this statement. Instead, they identify constructs such as these, and substitute the computed values at compile time (in this case, 2,048,000), usually in the intermediate representation (IR) tree.

In some compilers, constant folding is done early so that statements such as C's array initializers can accept simple arithmetic expressions. However, it is also common to include further constant folding rounds in later stages in the compiler, as well.

Constant folding can be done in a compiler's front end on the IR tree that represents the high-level source language, before it is translated into three-address code, or in the back end, as an adjunct to constant propagation.

Constant folding and cross compilation

In implementing a cross compiler, care must be taken to ensure that the behaviour of the arithmetic operations on the host architecture matches that on the target architecture, as otherwise enabling constant folding will change the behaviour of the program. This is of particular importance in the case of floating point operations, whose precise implementation may vary widely.

Constant propagation

Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values. Consider the following pseudocode:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Propagating x yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Continuing to propagate yields the following (which would likely be further optimized by dead code elimination of both x and y.)

```
int x = 14;
int y = 0;
```

```
return 0;
```

Constant propagation is implemented in compilers using reaching definition analysis results. If a variable's all reaching definitions are the same assignment which assigns a same constant to the variable, then the variable has a constant value and can be replaced with the constant.

Constant propagation can also cause conditional branches to simplify to one or more unconditional statements, when the conditional expression can be evaluated to true or false at compile time to determine the only possible outcome.

The optimizations in action

Constant folding and propagation are typically used together to achieve many simplifications and reductions, by interleaving them iteratively until no more changes occur. Consider this pseudocode, for example:

```
int a = 30;
int b = 9 - a / 5;
int c;

c = b * 4;
if (c > 10) {
    c = c - 10;
}
return c * (60 / a);
```

Applying constant propagation once, followed by constant folding, yields:

```
int a = 30;
int b = 3;
int c;

c = 3 * 4;
if (c > 10) {
    c = c - 10;
}
return c * 2;
```

As `a` and `b` have been simplified to constants and their values substituted everywhere they occurred, the compiler now applies dead code elimination to discard them, reducing the code further:

```
int c;

c = 12;
if (12 > 10) {
    c = 2;
}
return c * 2;
```

The compiler can now detect that the `if` statement will always evaluate to true, `c` itself can be eliminated, shrinking the code even further:

```
return 4;
```

If this pseudocode constituted the body of a function, the compiler could further take advantage of the knowledge that it evaluates to the constant integer 4 to eliminate unnecessary calls to the function, producing further performance gains.

Further reading

- Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann. 1997.

Sparse conditional constant propagation

In computer science, **sparse conditional constant propagation** is an optimization frequently applied in compilers after conversion to static single assignment form (SSA). It simultaneously removes some kinds of dead code and propagates constants throughout a program. However, it is strictly more powerful than applying dead code elimination and constant propagation in any order or any number of repetitions.^[1] ^[2]

The algorithm operates by performing abstract interpretation of the code in SSA form. During abstract interpretation, it typically uses a flat lattice of constants for values and a global environment mapping SSA variables to values into this lattice. The crux of the algorithm comes in how it handles the interpretation of branch instructions. When encountered, the condition for a branch is evaluated *as best as possible* given the precision of the abstract values bound to variables in the condition. It may be the case that the values are perfectly precise (neither top nor bottom) and hence, abstract execution can decide in which direction to branch. If the values are not constant, or a variable in the condition is undefined, then both branch directions must be taken to remain conservative.

Upon completion of the abstract interpretation, instructions which were never reached are marked as dead code. SSA variables found to have constant values may then be inlined at (propagated to) their point of use.

Notes

[1] Wegman, Mark N. and Zadeck, F. Kenneth. "Constant Propagation with Conditional Branches." *ACM Transactions on Programming Languages and Systems*, 13(2), April 1991, pages 181-210.

[2] Click, Clifford and Cooper, Keith. "Combining Analyses, Combining Optimizations", *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995, pages 181-196

References

- Cooper, Keith D. and Torczon, Linda. *Engineering a Compiler*. Morgan Kaufmann. 2005.

Common subexpression elimination

In computer science, **common subexpression elimination** (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyses whether it is worthwhile replacing them with a single variable holding the computed value.

Example

In the following code:

```
a = b * c + g;  
d = b * c * d;
```

it may be worth transforming the code so that it is translated as if it had been written:

```
tmp = b * c;  
a = tmp + g;  
d = tmp * d;
```

"worth" means that the resulting code would execute faster.

Principle

The possibility to perform CSE is based on available expression analysis (a data flow analysis). An expression $b*c$ is available at a point p in a program if:

- every path from the initial node to p evaluates $b*c$ before reaching p ,
- and there are no assignments to b or c after the evaluation but before p .

The cost/benefit analysis performed by an optimizer will calculate whether the cost of the store to `tmp` is less than the cost of the multiplication; in practice other factors such as which values are held in which registers are also significant.

Compiler writers distinguish two kinds of CSE:

- **local common subexpression elimination** works within a single basic block
- **global common subexpression elimination** works on an entire procedure,

Both kinds rely on data flow analysis of which expressions are available at which points in a program.

Benefits

The benefits of performing CSE are great enough that it is a commonly used optimization.

In simple cases like in the example above, programmers may manually eliminate the duplicate expressions while writing the code. The greatest source of CSEs are intermediate code sequences generated by the compiler, such as for array indexing calculations, where it is not possible for the developer to manually intervene. In some cases language features may create many duplicate expressions. For instance, C macros, where macro expansions may result in common subexpressions not apparent in the original source code.

Compilers need to be judicious about the number of temporaries created to hold values. An excessive number of temporary values creates register pressure possibly resulting in spilling registers to memory, which may take longer than simply recomputing an arithmetic result when it is needed.

References

- Steven S. Muchnick, *Advanced Compiler Design and Implementation* (Morgan Kaufmann, 1997) pp. 378-396
- John Cocke. "Global Common Subexpression Elimination." *Proceedings of a Symposium on Compiler Construction, ACM SIGPLAN Notices* 5(7), July 1970, pages 850-856.
- Briggs, Preston, Cooper, Keith D., and Simpson, L. Taylor. "Value Numbering." *Software-Practice and Experience*, 27(6), June 1997, pages 701-724.

Partial redundancy elimination

In compiler theory, **partial redundancy elimination** (PRE) is a compiler optimization that eliminates expressions that are redundant on some but not necessarily all paths through a program. PRE is a form of common subexpression elimination.

An expression is called partially redundant if the value computed by the expression is already available on some but not all paths through a program to that expression. An expression is fully redundant if the value computed by the expression is available on all paths through the program to that expression. PRE can eliminate partially redundant expressions by inserting the partially redundant expression on the paths that do not already compute it, thereby making the partially redundant expression fully redundant.

For instance, in the following code:

```
if (some_condition) {
    // some code
    y = x + 4;
}
else {
    // other code
}
z = x + 4;
```

the expression `x+4` assigned to `z` is partially redundant because it is computed twice if `some_condition` is true. PRE would perform code motion on the expression to yield the following optimized code:

```
if (some_condition) {
    // some code
    t = x + 4;
    y = t;
}
else {
    // other code
    t = x + 4;
}
z = t;
```

An interesting property of PRE is that it performs (a form of) common subexpression elimination and loop-invariant code motion at the same time. In addition, PRE can be extended to eliminate *injured* partial redundancies, thereby effectively performing strength reduction. This makes PRE one of the most important optimizations in optimizing compilers. Traditionally, PRE is applied to lexically equivalent expressions, but recently formulations of PRE based on static single assignment form have been published that apply the PRE algorithm to values instead of expressions,

unifying PRE and global value numbering.

Further reading

- Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann. 1997.
- Morel, E., and Renvoise, C. *Global Optimization by Suppression of Partial Redundancies*. Communications of the acm, Vol. 22, Num. 2, Feb. 1979.
- Knoop, J., Ruthing, O., and Steffen, B. *Lazy Code Motion*. ACM SIGPLAN Notices Vol. 27, Num. 7, Jul. 1992, '92 Conference on PLDI.
- Paleri, V. K., Srikant, Y. N., and Shankar, P. *A Simple Algorithm for Partial Redundancy Elimination*. SIGPLAN Notices, Vol. 33(12). pages 35–43 (1998).
- Kennedy, R., Chan, S., Liu, S.M., Lo, R., Peng, T., and Chow, F. *Partial Redundancy Elimination in SSA Form*. ACM Transactions on Programming Languages Vol. 21, Num. 3, pp. 627-676, 1999.
- VanDrunen, T., and Hosking, A.L. *Value-Based Partial Redundancy Elimination*, Lecture Notes in Computer Science Vol. 2985/2004, pp. 167 - 184, 2004.
- Xue, J. and Knoop, J. *A Fresh Look at PRE as a Maximum Flow Problem*. International Conference on Compiler Construction (CC'06), pages 139—154, Vienna, Austria, 2006.
- Xue, J. and Cai Q. *A lifetime optimal algorithm for speculative PRE*. ACM Transactions on Architecture and Code Optimization Vol. 3, Num. 3, pp. 115-155, 2006.

Global value numbering

Global value numbering (GVN) is a compiler optimization based on the SSA intermediate representation. It sometimes helps eliminate redundant code that common subexpression elimination (CSE) does not. At the same time, however, CSE may eliminate code that GVN does not, so both are often found in modern compilers. Global value numbering is distinct from local value numbering in that the value-number mappings hold across basic block boundaries as well, and different algorithms are used to compute the mappings.

Global value numbering works by assigning a value number to variables and expressions. To those variables and expressions which are provably equivalent, the same value number is assigned. For instance, in the following code:

```
w := 3
x := 3
y := x + 4
z := w + 4
```

a good GVN routine would assign the same value number to `w` and `x`, and the same value number to `y` and `z`. For instance, the map $[w \mapsto 1, x \mapsto 1, y \mapsto 2, z \mapsto 2]$ would constitute an optimal value-number mapping for this block. Using this information, the previous code fragment may be safely transformed into:

```
w := 3
x := w
y := w + 4
z := y
```

Depending on the code following this fragment, copy propagation may be able to remove the assignments to `x` and to `z`.

The reason that GVN is sometimes more powerful than CSE comes from the fact that CSE matches lexically identical expressions whereas the GVN tries to determine an underlying equivalence. For instance, in the code:

```
a := c * d
e := c
f := e * d
```

CSE would *not* eliminate the recomputation assigned to `f`, but even a poor GVN algorithm should discover and eliminate this redundancy.

SSA form is required to perform GVN so that false variable name-value name mappings are not created.

References

- Alpern, Bowen, Wegman, Mark N., and Zadeck, F. Kenneth. "Detecting Equality of Variables in Programs.", *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (POPL), ACM Press, San Diego, CA, USA, January 1988, pages 1–11.
- L. Taylor Simpson, "Value-Driven Redundancy Elimination." Technical Report 96-308, Computer Science Department, Rice University, 1996. (Author's Ph.D. thesis)
- Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann. 1997.

Strength reduction

Strength reduction is a compiler optimization where expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. (Cooper, Simpson & Vick 1995, p. 1)

Examples of strength reduction include:

- replacing a multiplication within a loop with an addition
- replacing an exponentiation within a loop with a multiplication

Code analysis

Most of a program's execution time is typically spent in a small section of code, and that code is often inside a loop that is executed over and over.

A compiler uses methods to identify loops and recognize the characteristics of register values within those loops. For strength reduction, the compiler is interested in

- loop invariants. These are values that do not change within the body of a loop.
- induction variables. These are the values that are being iterated each time through the loop.

Loop invariants are essentially constants within a loop, but their value may change outside of the loop. Induction variables are changing by known amounts. The terms are relative to a particular loop. When loops are nested, an induction variable in the outer loop can be a loop invariant in the inner loop.

Strength reduction looks for expressions involving a loop invariant and an induction variable. Some of those expressions can be simplified. For example, the multiplication of loop invariant `c` and induction variable `i`

```
c = 8;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
```

can be replaced with successive weaker additions

```
c = 8;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

Strength reduction example

Below is an example that will strength reduce all the loop multiplications that arose from array indexing address calculations.

Imagine a simple loop that sets an array to the identity matrix.

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        A[i, j] = 0.0;
    }
    A[i, i] = 1.0;
}
```

Intermediate Code

The compiler will view this code as

```
0010 // for (i = 0, i < n; i++)
0020     {
0030         r1 = #0                                // i = 0
0040     G0000:
0050         load r2, n                            // i < n
0060         cmp r1, r2
0070         bge G0001
0080
0090     // for (j = 0; j < n; j++)
0100         {
0110             r3    = #0                                // j = 0
0120     G0002:
0130         load r4, n                            // j < n
0140         cmp r3, r4
0150         bge G0003
0160
0170         // A[i, j] = 0.0;
0180         load r7, n
0190         r8    = r1 * r7                          // calculate subscript i * n + j
0200         r9    = r8 + r3
0210         r10   = r9 * #8                         // calculate byte address
0220         fr3  = #0.0
```

```

0230          fstore fr3, A[r10]
0240
0250          r3 = r3 + #1           // j++
0260          br G0002
0270      }
0280 G0003:
0290      // A[i,i] = 1.0;
0300      load r12, n           // calculate subscript i * n + i
0310      r13 = r1 * r12
0320      r14 = r13 + r1
0330      r15 = r14 * #8        // calculate byte address
0340      fr4 = #1.0
0350      fstore fr4, A[r15]
0360
0370      //i++
0380      r1 = r1 + #1
0390      br G0000
0400  }
0410 G0001:

```

Many optimizations

The compiler will start doing many optimizations – not just strength reduction. Expressions that are constant (invariant) within a loop will be hoisted out of the loop. Constants can be loaded outside of both loops—such as floating point registers fr3 and fr4. Recognition that some variables don't change allow registers to be merged; n is constant, so r2, r4, r7, r12 can be hoisted and collapsed. The common value $i \cdot n$ is computed in (the hoisted) r8 and r13, so they collapse. The innermost loop (0120-0260) has been reduced from 11 to 7 intermediate instructions. The only multiply that remains in the innermost loop is line 0210's multiply by 8.

```

0010 // for (i = 0, i < n; i++)
0020 {
0030     r1 = #0                  // i = 0
0050     load r2, n
0130 //     load r4, n    killed; use r2
0180 //     load r7, n    killed; use r2
0300 //     load r12, n   killed; use r2
0220     fr3 = #0.0
0340     fr4 = #1.0
0040 G0000:
0060     cmp r1, r2           // i < n
0070     bge G0001
0080
0190     r8 = r1 * r2         // calculate subscript i * n
0310 //     r13 = r1 * r2    killed; use r8 // calculate subscript i * n
0090     // for (j = 0; j < n; j++)
0100     {
0110         r3 = #0           // j = 0
0120 G0002:
0140     cmp r3, r2           // j < n

```

```

0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0200      r9 = r8 + r3          // calculate subscript i * n + j
0210      r10 = r9 * #8         // calculate byte address
0230      fstore fr3, A[r10]
0240
0250      r3 = r3 + #1          // j++
0260      br G0002
0270      }
0280 G0003:
0290      // A[i,i] = 1.0;
0320      r14 = r8 + r1          // calculate subscript i * n + i
0330      r15 = r14 * #8         // calculate byte address
0350      fstore fr4, A[r15]
0360
0370      //i++
0380      r1 = r1 + #1
0390      br G0000
0400      }
0410 G0001:

```

There are more optimizations to go. Register r3 is the main variable in the innermost loop (0140-0260); it gets incremented by 1 each time through the loop. Register r8 (which is invariant in the innermost loop) is added to r3. Instead of using r3, the compiler can eliminate r3 and use r9. The loop, instead of being controlled by $r3 = 0$ to $n-1$, it can be controlled by $r9=r8+0$ to $r8+n-1$. That adds four instructions and kills four instructions, but there's one fewer instructions inside the loop

```

0110 //      r3    = #0      killed      // j = 0
0115      r9    = r8          // new assignment
0117      r20   = r8 + r2      // new limit
0120 G0002:
0140 //      cmp r3, r2      killed      // j < n
0145      cmp r9, r20        // r8 + j < r8 + n = r20
0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0200 //      r9    = r8 + r3 killed      // calculate subscript i * n + j
0210      r10   = r9 * #8       // calculate byte address
0230      fstore fr3, A[r10]
0240
0250 //      r3    = r3 + #1  killed      // j++
0255      r9    = r9 + #1       // new loop variable
0260      br G0002

```

Now r9 is the loop variable, but it interacts with the multiply by 8. Here we get to do some strength reduction. The multiply by 8 can be reduced to some successive additions of 8. Now there are no multiplications inside the loop.

```

0115      r9    = r8          // new assignment
0117      r20   = r8 + r2    // new limit
0118      r10   = r8 * #8    // initial value of r10
0120  G0002:
0145      cmp  r9, r20      // r8 + j < r8 + n = r20
0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0210  //      r10 = r9 * #8    killed // calculate byte address
0230      fstore fr3, A[r10]
0240
0245      r10 = r10 + #8    // strength reduced multiply
0255      r9 = r9 + #1      // loop variable
0260      br  G0002

```

Registers r9 and r10 ($= 8 \times r9$) aren't both needed; r9 can be eliminated in the loop. The loop is now 5 instructions.

```

0115  //      r9    = r8      killed
0117      r20   = r8 + r2    // limit
0118      r10   = r8 * #8    // initial value of r10
0119      r22   = r20 * #8    // new limit
0120  G0002:
0145  //      cmp  r9, r20    killed // r8 + j < r8 + n = r20
0147      cmp  r10, r22      // r10 = 8*(r8 + j) < 8*(r8 + n) = r22
0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0230      fstore fr3, A[r10]
0240
0245      r10 = r10 + #8    // strength reduced multiply
0255  //      r9 = r9 + #1    killed // loop variable
0260      br  G0002

```

Outer loop

Back to the whole picture:

```

0010 // for (i = 0, i < n; i++)
0020 {
0030     r1 = #0          // i = 0
0050     load r2, n
0220     fr3 = #0.0
0340     fr4 = #1.0
0040  G0000:
0060     cmp  r1, r2      // i < n
0070     bge G0001
0080
0190     r8    = r1 * r2    // calculate subscript i * n
0117     r20   = r8 + r2    // limit
0118     r10   = r8 * #8    // initial value of r10

```

```

0119      r22 = r20 * #8           // new limit
0090      // for (j = 0; j < n; j++)
0100      {
0120 G0002:
0147      cmp r10, r22           // r10 = 8*(r8 + j) < 8*(r8 + n) = r22
0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0230      fstore fr3, A[r10]
0240
0245      r10 = r10 + #8         // strength reduced multiply
0260      br G0002
0270      }
0280 G0003:
0290      // A[i,i] = 1.0;
0320      r14 = r8 + r1           // calculate subscript i * n + i
0330      r15 = r14 * #8          // calculate byte address
0350      fstore fr4, A[r15]
0360
0370      //i++
0380      r1 = r1 + #1
0390      br G0000
0400      }
0410 G0001:

```

There are now four multiplications within the outer loop that increments r1. Register $r8 = r1 * r2$ at 0190 can be strength reduced by setting it before entering the loop (0055) and incrementing it by $r2$ at the bottom of the loop (0385).

The value $r8 * 8$ (at 0118) can strength reduced by initializing it (0056) and adding $8 * r2$ to it when $r8$ gets incremented (0386).

Register $r20$ is being incremented by the invariant/constant $r2$ each time through the loop at 0117. After being incremented, it is multiplied by 8 to create $r22$ at 0119. That multiplication can be strength reduced by adding $8 * r2$ each time through the loop.

```

0010 // for (i = 0, i < n; i++)
0020 {
0030     r1 = #0                   // i = 0
0050     load r2, n
0220     fr3 = #0.0
0340     fr4 = #1.0
0055     r8 = r1 * r2           // set initial value for r8
0056     r40 = r8 * #8            // initial value for r8 * 8
0057     r30 = r2 * #8            // increment for r40
0058     r20 = r8 + r2            // copied from 0117
0058     r22 = r20 * #8            // initial value of r22
0040 G0000:
0060     cmp r1, r2                // i < n
0070     bge G0001

```

```

0080
0190 // r8    = r1 * r2      killed      // calculate subscript i * n
0117 // r20   = r8 + r2      killed - dead code
0118 r10   = r40                  // strength reduced expression to r40
0119 // r22   = r20 * #8      killed      // strength reduced
0090 // for (j = 0; j < n; j++)
0100 {
0120 G0002:
0147     cmp r10, r22          // r10 = 8*(r8 + j) < 8*(r8 + n) = r22
0150     bge G0003
0160
0170     // A[i,j] = 0.0;
0230     fstore fr3, A[r10]
0240
0245     r10 = r10 + #8        // strength reduced multiply
0260     br G0002
0270 }
0280 G0003:
0290     // A[i,i] = 1.0;
0320     r14 = r8 + r1          // calculate subscript i * n + i
0330     r15 = r14 * #8        // calculate byte address
0350     fstore fr4, A[r15]
0360
0370     //i++
0380     r1 = r1 + #1
0385     r8 = r8 + r2          // strength reduce r8 = r1 * r2
0386     r40 = r40 + r30        // strength reduce expression r8 * 8
0388     r22 = r22 + r30        // strength reduce r22 = r20 * 8
0390     br G0000
0400 }
0410 G0001:

```

The last multiply

That leaves the two loops with only one multiplication operation (at 0330) within the outer loop and no multiplications within the inner loop.

```

0010 // for (i = 0, i < n; i++)
0020 {
0030     r1 = #0                  // i = 0
0050     load r2, n
0220     fr3 = #0.0
0340     fr4 = #1.0
0055     r8 = r1 * r2          // set initial value for r8
0056     r40 = r8 * #8         // initial value for r8 * 8
0057     r30 = r2 * #8         // increment for r40
0058     r20 = r8 + r2          // copied from 0117
0058     r22 = r20 * #8         // initial value of r22
0040 G0000:

```

```

0060      cmp r1, r2                      // i < n
0070      bge G0001
0080
0118      r10 = r40                      // strength reduced expression to r40
0090      // for (j = 0; j < n; j++)
0100      {
0120  G0002:
0147      cmp r10, r22                     // r10 = 8*(r8 + j) < 8*(r8 + n) = r22
0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0230      fstore fr3, A[r10]
0240
0245      r10 = r10 + #8                  // strength reduced multiply
0260      br G0002
0270      }
0280  G0003:
0290      // A[i,i] = 1.0;
0320      r14 = r8 + r1                  // calculate subscript i * n + i
0330      r15 = r14 * #8                // calculate byte address
0350      fstore fr4, A[r15]
0360
0370      //i++
0380      r1 = r1 + #1
0385      r8 = r8 + r2                  // strength reduce r8 = r1 * r2
0386      r40 = r40 + r30                // strength reduce expression r8 * 8
0388      r22 = r22 + r30                // strength reduce r22 = r20 * 8
0390      br G0000
0400      }
0410  G0001:

```

At line 0320, r14 is the sum of r8 and r1, and r8 and r1 are being incremented in the loop. Register r8 is being bumped by r2 ($=n$) and r1 is being bumped by 1. Consequently, r14 is being bumped by $n+1$ each time through the loop. The last loop multiply at 0330 can be strength reduced by adding $(r2+1)*8$ each time through the loop.

```

0010 // for (i = 0, i < n; i++)
0020 {
0030     r1 = #0                         // i = 0
0050     load r2, n
0220     fr3 = #0.0
0340     fr4 = #1.0
0055     r8 = r1 * r2                  // set initial value for r8
0056     r40 = r8 * #8                // initial value for r8 * 8
0057     r30 = r2 * #8                // increment for r40
0058     r20 = r8 + r2                  // copied from 0117
0058     r22 = r20 * #8                // initial value of r22
005A     r14 = r8 + #1                // copied from 0320
005B     r15 = r14 * #8                // initial value of r15 (0330)

```

```

005C      r49 = r2 + 1
005D      r50 = r49 * #8          // strength reduced increment
0040  G0000:
0060      cmp r1, r2            // i < n
0070      bge G0001
0080
0118      r10 = r40            // strength reduced expression to r40
0090      // for (j = 0; j < n; j++)
0100      {
0120  G0002:
0147      cmp r10, r22          // r10 = 8*(r8 + j) < 8*(r8 + n) = r22
0150      bge G0003
0160
0170      // A[i,j] = 0.0;
0230      fstore fr3, A[r10]
0240
0245      r10 = r10 + #8        // strength reduced multiply
0260      br G0002
0270      }
0280  G0003:
0290      // A[i,i] = 1.0;
0320  // r14 = r8 + r1      killed // dead code
0330  // r15 = r14 * #8      killed // strength reduced
0350      fstore fr4, A[r15]
0360
0370      //i++
0380      r1 = r1 + #1
0385      r8 = r8 + r2          // strength reduce r8 = r1 * r2
0386      r40 = r40 + r30        // strength reduce expression r8 * 8
0388      r22 = r22 + r30        // strength reduce r22 = r20 * 8
0389      r15 = r15 + r50        // strength reduce r15 = r14 * 8
0390      br G0000
0400      }
0410  G0001:

```

There's still more to go. Constant folding will recognize that $r1=0$ in the preamble, so several instruction will clean up. Register $r8$ isn't used in the loop, so it can disappear.

Furthermore, $r1$ is only being used to control the loop, so $r1$ can be replaced by a different induction variable such as $r40$. Where i went $0 \leq i < n$, register $r40$ goes $0 \leq r40 < 8 * n * n$.

```

0010 // for (i = 0, i < n; i++)
0020  {
0030  // r1 = #0                // i = 0, becomes dead code
0050  load r2, n
0220  fr3 = #0.0
0340  fr4 = #1.0
0055  // r8 = #0      killed // r8 no longer used
0056  r40 = #0            // initial value for r8 * 8

```

```
0057      r30 = r2 * #8           // increment for r40
0058 // r20 = r2                 killed   // r8 = 0, becomes dead code
0058      r22 = r2 * #8           // r20 = r2
005A // r14 =      #1    killed   // r8 = 0, becomes dead code
005B      r15 =      #8           // r14 = 1
005C      r49 = r2 + 1
005D      r50 = r49 * #8           // strength reduced increment
005D      r60 = r2 * r30          // new limit for r40
0040 G0000:
0060 // cmp r1, r2      killed   // i < n; induction variable replaced
0065      cmp r40, r60          // i * 8 * n < 8 * n * n
0070      bge G0001
0080
0118      r10 = r40             // strength reduced expression to r40
0090 // for (j = 0; j < n; j++)
0100 {
0120 G0002:
0147      cmp r10, r22          // r10 = 8*(r8 + j) < 8*(r8 + n) = r22
0150      bge G0003
0160
0170 // A[i,j] = 0.0;
0230      fstore fr3, A[r10]
0240
0245      r10 = r10 + #8           // strength reduced multiply
0260      br G0002
0270 }
0280 G0003:
0290 // A[i,i] = 1.0;
0350      fstore fr4, A[r15]
0360
0370 //i++
0380 // r1 = r1 + #1      killed   // dead code (r40 controls loop)
0385 // r8 = r8 + r2      killed   // dead code
0386      r40 = r40 + r30          // strength reduce expression r8 * 8
0388      r22 = r22 + r30          // strength reduce r22 = r20 * 8
0389      r15 = r15 + r50          // strength reduce r15 = r14 * 8
0390      br G0000
0400 }
0410 G0001:
```

Other strength reduction operations

This material is disputed. It is better described as peephole optimizations and instruction assignment.

Operator strength reduction uses mathematical identities to replace slow math operations with faster operations. The benefits depend on the target CPU and sometimes on the surrounding code (which can affect the availability of other functional units within the CPU).

- replacing integer division or multiplication by a power of 2 with an arithmetic shift or logical shift^[1]
- replacing integer multiplication by a constant with a combination of shifts, adds or subtracts

original calculation	replacement calculation
$y = x / 8$	$y = x \gg 3$
$y = x * 64$	$y = x \ll 6$
$y = x * 2$	$y = x + x$
$y = x * 15$	$y = (x \ll 4) - x$

Induction variable (orphan)

Induction variable or recursive strength reduction replaces a function of some systematically changing variable with a simpler calculation using previous values of the function. In a procedural programming language this would apply to an expression involving a loop variable and in a declarative language it would apply to the argument of a recursive function. For example,

```
f x = ... (2 ** x) ... (f (x + 1)) ...
```

becomes

```
f x = f' x 1
where
f' x z = ... z ... (f' (x + 1) (2 * z)) ...
```

Here the expensive operation $(2 ** x)$ has been replaced by the cheaper $(2 * z)$ in the recursive function f . This maintains the invariant that $z = 2 ** x$ for any call to f' .

Notes

[1] In languages such as C and Java, integer division has round-towards-zero semantics, whereas a bit-shift always rounds down, requiring special treatment for negative numbers. For example, in Java, $-3 / 2$ evaluates to -1 , whereas $-3 \gg 1$ evaluates to -2 . So in this case, the compiler *cannot* optimize division by two by replacing it by a bit shift.

References

- Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986), *Compilers: Principles, Techniques, and Tools* (2nd ed.), ISBN 978-0201100884
- Allen, Francis E.; Cocke, John; Kennedy, Ken (1981), "Reduction of Operator Strength", in Munchnik, Steven S.; Jones, Neil D., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, ISBN 978-0137296811
- Cocke, John; Kennedy, Ken (November 1977), "An algorithm for reduction of operator strength", *Communications of the ACM* **20** (11)
- Cooper, Keith; Simpson, Taylor; Vick, Christopher (October 1995), *Operator Strength Reduction* (<http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR95635-S.pdf>), Rice University, retrieved April 22, 2010

Bounds-checking elimination

In computer science, **bounds-checking elimination** is a compiler optimization useful in programming languages or runtimes that enforce bounds checking, the practice of consistently checking every index into an array to verify that the index is within the defined valid range of indexes. Its goal is to detect which of these indexing operations do not need to be validated at runtime, and eliminating those checks.

One common example is accessing an array element, modifying it, and storing the modified value in the same array at the same location. Normally, this example would result in a bounds check when the element is read from the array and a second bounds check when the modified element is stored using the same array index. Bounds-checking elimination could eliminate the second check if the compiler or runtime can determine that the array size cannot change and the index cannot change between the two array operations.

Another example occurs when a programmer loops over the elements of the array, and the loop condition guarantees that the index is within the bounds of the array. It may be difficult to detect that the programmer's manual check renders the automatic check redundant. However, it may still be possible for the compiler or runtime to perform proper bounds-checking elimination in this case.

Implementations

In natively compiled languages

One technique for bounds-checking elimination is to use a typed static single assignment form representation and for each array create a new type representing a safe index for that particular array. The first use of a value as an array index results in a runtime type cast (and appropriate check), but subsequently the safe index value can be used without a type cast, without sacrificing correctness or safety.

In JIT-compiled languages

Just-in-time compiled languages such as Java often check indexes at runtime before accessing Arrays. Some just-in-time compilers such as HotSpot are able to eliminate some of these checks if they discover that the index is always within the correct range, or if an earlier check would have already thrown an exception^[1] [2].

References

- [1] Kawaguchi, Kohsuke (2008-03-30). "Deep dive into assembly code from Java" (http://weblogs.java.net/blog/kohsuke/archive/2008/03/deep_dive_into.html). . Retrieved 2008-04-02.
- [2] "Fast, Effective Code Generation in a Just-In-Time Java Compiler" (<http://ei.cs.vt.edu/~cs5314/presentations/Group2PLDI.pdf>). Intel Corporation. . Retrieved 2007-06-22.

External links

- W. Amme, J. von Ronne, M. Franz. Using the SafeTSA Representation to Boost the Performance of an Existing Java Virtual Machine (<http://citeseer.ist.psu.edu/721276.html>) (2002).

Inline expansion

In computing, **inline expansion**, or **inlining**, is a manual or compiler optimization that replaces a function call site with the body of the callee. This optimization may improve time and space usage at runtime, at the possible cost of increasing the final size of the program (i.e. the binary file size).

Ordinarily, when a function is invoked, control is transferred to its definition by a branch or call instruction. With inlining, control drops through directly to the code for the function, without a branch or call instruction. Inlining improves performance in several ways:

- It removes the cost of the function call and return instructions, as well as any other prologue and epilog code injected into every function by the compiler.
- Eliminating branches and keeping code that is executed close together in memory improves instruction cache performance by improving locality of reference.
- Once inlining has been performed, additional intraprocedural optimizations become possible on the inlined function body. For example, a constant passed as an argument, can often be propagated to all instances of the matching parameter, or part of the function may be "hoisted out" of a loop.

The primary cost of inlining is that it tends to increase code size, although it does not always do so. Inlining may also decrease performance in some cases - for instance, multiple copies of a function may increase code size enough that the code no longer fits in the cache, resulting in more cache misses.

Some languages (for example, C and C++) support the `inline` keyword in function definitions. This keyword serves as a "hint" to the compiler that it should try to inline the function. Compilers use a variety of mechanisms, including hints from programmers, to decide which function calls should be inlined.

Compilers usually implement statements with inlining. Loop conditions and loop bodies need lazy evaluation. This property is fulfilled when the code to compute loop conditions and loop bodies is inlined. Performance considerations are another reason to inline statements.

In the context of functional programming languages, inline expansion is usually followed by the beta-reduction transformation.

A programmer might inline a function manually through copy and paste programming, as a one-time operation on the source code. However, other methods of controlling inlining (see below) are preferable, because they do not precipitate bugs arising when the programmer overlooks a (possibly modified) duplicated version of the original function body, while fixing a bug in the inlined function.

Implementation

Once the compiler has decided to inline a particular function, performing the inlining operation itself is usually simple. Depending on whether the compiler inlines functions across code in different languages, the compiler can do inlining on either a high-level intermediate representation (like abstract syntax trees) or a low-level intermediate representation. In either case, the compiler simply computes the arguments, stores them in variables corresponding to the function's arguments, and then inserts the body of the function at the call site.

Linkers, as well as compilers, can also do function inlining. When a linker inlines functions, it may inline functions whose source is not available, such as library functions (see link-time optimization). A run-time system can inline function as well. Run-time inlining can use dynamic profiling information to make better decisions about which functions to inline, as in the Java Hotspot compiler.

Here is a simple example of inline expansion performed "by hand" at the source level in the C programming language:

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}
```

Before inlining:

```
int f(int y) {
    return pred(y) + pred(0) + pred(y+1);
}
```

After inlining:

```
int f(int y) {
    int temp = 0;
    if (y == 0) temp += 0; else temp += y - 1; /* (1) */
    if (0 == 0) temp += 0; else temp += 0 - 1; /* (2) */
    if (y+1 == 0) temp += 0; else temp += (y + 1) - 1; /* (3) */
    return temp;
}
```

Note that this is only an example. In an actual C application, it would be preferable to use an inlining language feature such as parameterized macros or inline functions to tell the compiler to transform the code in this way. The next section lists ways to optimize this code.

Inlining by Assembly macro expansion

Assembler macros provide an alternative approach to inlining whereby a sequence of instructions can normally be generated inline by macro expansion from a single macro source statement (with zero or more parameters).^[1] One of the parameters might be an option to alternatively generate a one-time separate subroutine containing the sequence and processed instead by an inlined call to the function. Example:

```
MOVE FROM=array1, TO=array2, INLINE=NO
```

Benefits

Inline expansion itself is an optimization, since it eliminates overhead from calls, but it is much more important as an enabling transformation. That is, once the compiler expands a function body in the context of its call site—often with arguments that may be fixed constants -- it may be able to do a variety of transformations that were not possible before. For example, a conditional branch may turn out to be always true or always false at this particular call site. This in turn may enable dead code elimination, loop-invariant code motion, or induction variable elimination.

In the C example in the previous section, optimization opportunities abound. The compiler may follow this sequence of steps:

- The `temp += 0` statements in the lines marked (1), (2) and (3) do nothing. The compiler can remove them.
- The condition `0 == 0` is always true, so the compiler can replace the line marked (2) with the consequent, `temp += 0` (which does nothing).
- The compiler can rewrite the condition `y+1 == 0` to `y == -1`.
- The compiler can reduce the expression `(y + 1) - 1` to `y` (assuming wraparound overflow semantics)
- The expressions `y` and `y+1` cannot both equal zero. This lets the compiler eliminate one test.

The new function looks like:

```
int f(int y) {
    if (y == 0)
        return y;           /* or return 0 */
    else if (y == -1)
        return y - 1;      /* or return -2 */
    else
        return y + y - 1;
}
```

Problems

Replacing a call site with an expanded function body can worsen performance in several ways :

- In applications where code size is more important than speed, such as many embedded systems, inlining is usually disadvantageous except for very small functions, such as trivial mutator methods.
- The increase in code size may cause a small, critical section of code to no longer fit in the cache, causing cache misses and slowdown.
- The added variables from the inlined procedure may consume additional registers, and in an area where register pressure is already high this may force spilling, which causes additional RAM accesses.
- A language specification may allow a program to make additional assumptions about arguments to procedures that it can no longer make after the procedure is inlined.
- If code size is increased too much, resource constraints such as RAM size may be exceeded, leading to programs that either cannot be run or that cause thrashing. Today, this is unlikely to be an issue with desktop or server computers except with very aggressive inlining, but it can still be an issue for embedded systems.

Typically, compiler developers keep these issues in mind, and incorporate heuristics into their compilers that choose which functions to inline so as to improve performance, rather than worsening it, in most cases.

Limitations

It is not always possible to inline a subroutine. Consider the case of a subroutine that calls itself recursively until it receives a particular piece of input data from a peripheral. The compiler cannot generally determine when this process will end, so it would never finish inlining if it was designed to inline every single subroutine invocation. Thus, compilers for languages which support recursion must have restrictions on what they will automatically choose to inline.

Selection methods and language support

Many compilers aggressively inline functions wherever it is beneficial to do so. Although it can lead to larger executables, aggressive inlining has nevertheless become more and more desirable as memory capacity has increased faster than CPU speed. Inlining is a critical optimization in functional languages and object-oriented programming languages, which rely on it to provide enough context for their typically small functions to make classical optimizations effective.

External links

- "Eliminating Virtual Function Calls in C++ Programs" ^[2] by Gerald Aigner and Urs Hözle
- "Reducing Indirect Function Call Overhead In C++ Programs" ^[3] by Brad Calder and Dirk Grumwald
- "Secrets of the Glasgow Haskell Compiler Inliner" ^[4] by Simon Peyton Jones and Simon Marlow
- ALTO - A Link-Time Optimizer for the DEC Alpha ^[5]
- Article "Advanced techniques" ^[6] by John R. Levine
- Article "Inlining Semantics for Subroutines which are Recursive" ^[7] by Henry G. Baker
- Article "Whole Program Optimization with Visual C++ .NET" ^[8] by Brandon Bray

References

- [1] http://en.wikipedia.org/wiki/Assembly_language#Macros
- [2] <http://citeseer.ist.psu.edu/aigner96eliminating.html>
- [3] <http://citeseer.ist.psu.edu/calder94reducing.html>
- [4] <http://research.microsoft.com/~simonpj/Papers/inlining/>
- [5] <http://www.cs.arizona.edu/alto/Doc/alto.html>
- [6] <http://www.iecc.com/linker/linker11.html>
- [7] <http://home.pipeline.com/~hbaker1/Inlines.html>
- [8] <http://codeproject.com/tips/gloption.asp>

Return value optimization

Return value optimization, or simply **RVO**, is a compiler optimization technique that involves eliminating the temporary object created to hold a function's return value.^[1] In C++, it is particularly notable for being allowed to change the observable behaviour of the resulting program.^[2]

Summary

In general, the C++ standard allows a compiler to perform any optimization, as long as the resulting executable exhibits the same observable behaviour *as if* all the requirements of the standard have been fulfilled. This is commonly referred to as the *as-if rule*.^[3] The term *return value optimization* refers to a special clause in the C++ standard that allows an implementation to omit a copy operation resulting from a return statement, even if the copy constructor has side effects,^[4] something that is not permitted by the *as-if rule* alone.^[3]

The following example demonstrates a scenario where the implementation may eliminate one or both of the copies being made, even if the copy constructor has a visible side effect (printing text).^[4] The first copy that may be eliminated is the one where `C()` is copied into the function `f`'s return value. The second copy that may be eliminated is the copy of the temporary object returned by `f` to `obj`.

```
#include <iostream>

struct C {
    C() {}
    C(const C&) { std::cout << "A copy was made.\n"; }
};

C f() {
    return C();
}
```

```
int main() {
    std::cout << "Hello World!\n";
    C obj = f();
}
```

Depending upon the compiler, and that compiler's settings, the resulting program may display any of the following outputs:

```
Hello World!
A copy was made.
A copy was made.
```

```
Hello World!
A copy was made.
```

```
Hello World!
```

Background

Returning an object of builtin type from a function usually carries little to no overhead, since the object typically fits in a CPU register. Returning a larger object of class type may require more expensive copying from one memory location to another. To achieve this, an implementation may create a hidden object in the caller's stack frame, and pass the address of this object to the function. The function's return value is then copied into the hidden object.^[5] Thus, code such as this:

```
struct Data {
    char bytes[16];
};

Data f() {
    Data result = {};
    // generate result
    return result;
}

int main() {
    Data d = f();
}
```

May generate code equivalent to this:

```
struct Data {
    char bytes[16];
};

Data * f(Data * _hiddenAddress) {
    Data result = {};
    // copy result into hidden object
    *_hiddenAddress = result;
    return _hiddenAddress;
}
```

```
int main() {
    Data _hidden; // create hidden object
    Data d = *f(&_hidden); // copy the result into d
}
```

which causes the `Data` object to be copied twice.

In the early stages of the evolution of C++, the language's inability to efficiently return an object of class type from a function was considered a weakness.^[6] Around 1991, Walter Bright invented a technique to minimize copying, effectively replacing the hidden object and the named object inside the function with the object used to hold the result:^[7]

```
struct Data {
    char bytes[16];
};

void f(Data *p) {
    // generate result directly in *p
}

int main() {
    Data d;
    f(&d);
}
```

Bright implemented this optimization in his Zortech C++ compiler.^[6] This particular technique was later coined "Named return value optimization", referring to the fact that the copying of a named object is elided.^[7]

Compiler support

Return value optimization is supported on most compilers^[1] [8] [9]. There may be, however, circumstances where the compiler is unable to perform the optimization. One common case is when a function may return different named objects depending on the path of execution: ^[8] [10] [5]

```
#include <string>
std::string f(bool cond = false) {
    std::string first("first");
    std::string second("second");
    // the function may return one of two named objects
    // depending on its argument. RVO might not be applied
    return cond ? first : second;
}

int main() {
    std::string result = f();
}
```

References

- [1] Meyers, Scott (1996). *More Effective C++*. Addison Wesley.
- [2] Alexandrescu, Andrei (2003-02-01). "Move Constructors" (<http://www.ddj.com/cpp/184403855>). Dr. Dobbs Journal. . Retrieved 2009-03-25.
- [3] ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §1.9 Program execution [intro.execution]* para. 1
- [4] ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §12.8 Copying class objects [class.copy]* para. 15
- [5] Bulka, Dov; David Mayhew (2000). *Efficient C++*. Addison-Wesley. ISBN 0-201-37950-3.
- [6] Lippman, Stan. "The Name Return Value Optimization" (<http://blogs.msdn.com/slippman/archive/2004/02/03/66739.aspx>). Stan Lippman. . Retrieved 2009-03-23.
- [7] "Glossary D Programming Language 2.0" (<http://www.digitalmars.com/d/2.0/glossary.html>). Digital Mars. . Retrieved 2009-03-23.
- [8] Shoukry, Ayman B.. "Named Return Value Optimization in Visual C++ 2005" ([http://msdn.microsoft.com/en-us/library/ms364057\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms364057(VS.80).aspx)). Microsoft. . Retrieved 2009-03-20.
- [9] "Options Controlling C++ Dialect" (http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_2.html#SEC5). GCC. 2001-03-17. . Retrieved 2009-03-20.
- [10] Hinnant, Howard; et al. (2002-09-10). "N1377: A Proposal to Add Move Semantics Support to the C++ Language" (<http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1377.htm>). WG21. . Retrieved 2009-03-25.

Dead code

Dead code is a computer programming term for code in the source code of a program which is executed but whose result is never used in any other computation.^[1] ^[2] The execution of dead code wastes computation time as its results are never used.

While the result of a dead computation may never be used the dead code may raise exceptions or affect some global state, thus removal of such code may change the output of the program and introduce unintended bugs. Compiler optimizations are typically conservative in their approach to dead code removal if there is any ambiguity as to whether removal of the dead code will affect the program output.

Example

```
int f (int x, int y)
{
    int z=x+y;
    return x*y;
}
```

In the above example the sum of `x` and `y` is computed but never used. It is thus dead code and can be removed.

```
public void Method() {
    final boolean debug=false;

    if (debug) {
        //do something...
    }
}
```

In the above example "do something" is never executed, and so it is dead code. The Java compiler is smart enough to not compile it.

Analysis

Dead code elimination is a form of compiler optimization in which dead code is removed from a program. Dead code analysis can be performed using live variable analysis, a form of static code analysis and data flow analysis. This is in contrast to unreachable code analysis which is based on control flow analysis.

The dead code elimination technique is in the same class of optimizations as unreachable code elimination and redundant code elimination.

In large programming projects, it is sometimes difficult to recognize and eliminate dead code, particularly when entire modules become dead. Test scaffolding can make it appear that the code is still live, and at times, contract language can require delivery of the code even when the code is no longer relevant.^[3]

References

- [1] Debray, S. K., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (Mar. 2000), 378–415. (<http://doi.acm.org/10.1145/349214.349233>)
- [2] Appel, A. W. 1998. Modern Compiler Implementation in Java. Cambridge University Press.
- [3] Douglas W. Jones Dead Code Maintenance, Risks 8.19 (Feb. 1, 1989) (<http://catless.com/Risks/8.19.html#subj2>)

External links

- Dead Code Detector (DCD) simply finds never used code in your Java/JEE applications (<http://java.net/projects/dcd/pages/Home>)
- Comparisons of some Java Dead Code Detector (<http://java.net/projects/dcd/pages/Alternatives>)
- UCDetector (<http://www.ucdetector.org/>) Eclipse PlugIn to find dead java code

Dead code elimination

In compiler theory, **dead code elimination** is a compiler optimization to remove code which does not affect the program results. Removing such code has two benefits: it shrinks program size, an important consideration in some contexts, and it allows the running program to avoid executing irrelevant operations, which reduces its running time. **Dead code** includes code that can never be executed (**unreachable code**), and code that only affects **dead variables**, that is, variables that are irrelevant to the program.

Examples

Consider the following example written in C.

```
int foo (void)
{
    int a = 24;
    int b = 25; /* Assignment to dead variable */
    int c;
    c = a << 2;
    return c;
    b = 24; /* Unreachable code */
    return 0;
}
```

Because the return statement is executed unconditionally, no feasible execution path reaches the assignment to b. Thus, the assignment is **unreachable** and can be removed. (In a procedure with more complex control flow, such as

a label after the return statement and a `goto` elsewhere in the procedure, then a feasible execution path might exist through the assignment to `b`.)

Simple analysis of the uses of values would show that the value of `b` is not used inside `foo`. Furthermore, `b` is declared as a local variable inside `foo`, so its value cannot be used outside `foo`. Thus, the variable `b` is dead and an optimizer can reclaim its storage space and eliminate its initialization.

Also, even though some calculations are performed in the function, their values are not stored in locations accessible outside the scope of this function. Furthermore, given the function returns a static value (96), it may be simplified to the value it returns (this simplification is called Constant folding).

Most advanced compilers have options to activate dead code elimination, sometimes at varying levels. A lower level might only remove instructions that cannot be executed. A higher level might also not reserve space for unused variables. Yet a higher level might determine instructions or functions that serve no purpose and eliminate them.

A common use of dead code elimination is as an alternative to optional code inclusion via a preprocessor. Consider the following code.

```
int main(void) {
    int a = 5;
    int b = 6;
    int c;
    c = a * (b >> 1);
    if (0) { /* DEBUG */
        printf("%d\n", c);
    }
    return c;
}
```

Because the expression 0 will always evaluate to false, the code inside the if statement can never be executed, and dead code elimination would remove it entirely from the optimized program. This technique is common in debugging to optionally activate blocks of code; using an optimizer with dead code elimination eliminates the need for using a preprocessor to perform the same task.

In practice, much of the dead code that an optimizer finds is created by other transformations in the optimizer. For example, the classic techniques for operator strength reduction insert new computations into the code and render the older, more expensive computations dead.^[1] Subsequent dead code elimination removes those calculations and completes the effect (without complicating the strength-reduction algorithm).

Historically, dead code elimination was performed using information derived from data-flow analysis.^[2] An algorithm based on static single assignment form appears in the original journal article on SSA form by Cytron et al.^[3] Shillner improved on the algorithm and developed a companion algorithm for removing useless control-flow operations.^[4]

References

Partial dead code elimination using slicing transformations Found in: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation (PLDI '97) By Rastislav Bodík , Rajiv Gupta Issue Date:June 1997 pp. 682–694

- [1] Frances Allen, John Cocke, and Ken Kennedy. Reduction of Operator Strength. In *Program Flow Analysis*, Muchnick and Jones (editors), Prentice-Hall, 1981.
- [2] Ken Kennedy. A Survey of Data-flow Analysis Techniques. In *Program Flow Analysis*, Muchnick and Jones (editors), Prentice-Hall, 1981.
- [3] Ron Cytron, Jeanne Ferrante, Barry Rosen, and Ken Zadeck. Efficiently Computing Static Single Assignment Form and the Program Dependence Graph. ACM TOPLAS 13(4), 1991.
- [4] Keith D. Cooper and Linda Torczon, *Engineering a Compiler*, Morgan Kaufmann, 2003, pages 498ff.

Book references

- Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986). *Compilers - Principles, Techniques and Tools*. Addison Wesley Publishing Company. ISBN 0-201-10194-7.
- Muchnick, Steven S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers. ISBN 1-55860-320-4.
- Grune, Dick; Bal, Henri E.; Jacobs, Ceriel J.H.; Langendoen, Koen G. (2000). *Modern Compiler Design*. John Wiley & Sons, Inc.. ISBN 0-471-97697-0.

External Links

- How to trick C/C++ compilers into generating terrible code? (<http://www.futurechips.org/tips-for-power-coders/how-to-trick-cc-compilers-into-generating-terrible-code.html>)

Unreachable code

Unreachable code is a computer programming term for code in the source code of a program which can never be executed because there exists no control flow path to the code from the rest of the program.^[1]

Unreachable code is sometimes also called *dead code*, although dead code may also refer to code that is executed but has no effect on the output of a program.

Unreachable code is generally considered undesirable for a number of reasons, including:

- Occupies unnecessary memory
- Causes unnecessary caching of instructions into the CPU instruction cache - which also decreases data locality.
- From the perspective of program maintenance; time and effort may be spent maintaining and documenting a piece of code which is in fact unreachable, hence never executed.

Causes

The existence of unreachable code can be due to various factors, such as:

- complex conditional branches in which a case is never reachable;
- as a consequence of the internal transformations performed by an optimizing compiler;
- improper maintenance of a program or from debugging constructs and vestigial development code which have yet to be removed from a program.

In the latter case, code which is currently unreachable is there as part of a legacy. The distinguishing point in that case is that this part of code was once useful but is no longer used or required.

Examples

Consider the following fragment of C code:

```
int f (int x, int y)
{
    return x+y;
    int z=x*y;
}
```

The definition `int z=x*y;` is never reached as the function returns before the definition is reached. Therefore the definition of `z` can be discarded.

Analysis

Detecting unreachable code is a form of static analysis and involves performing control flow analysis to find any code that will never be executed regardless of the values of variables and other conditions at run time. In some languages (e.g. Java) some forms of unreachable code are explicitly disallowed. The optimization that removes unreachable code is known as dead code elimination.

Code may become *unreachable* as a consequence of the internal transformations performed by an optimizing compiler (e.g., common subexpression elimination).

In practice the sophistication of the analysis performed has a significant impact on the amount of unreachable code that is detected. For example, constant folding and simple flow analysis shows that the statement `xyz` in the following code is *unreachable*:

```
int n = 2 + 1;
if (n == 4)
{
    xyz
}
```

However, a great deal more sophistication is needed to work out that the statement `xyz` is unreachable in the following code:

```
double x = sqrt(2);
if (x > 5)
{
    xyz
}
```

The unreachable code elimination technique is in the same class of optimizations as dead code elimination and redundant code elimination.

Unreachability vs. profiling

In some cases, a practical approach may be a combination of simple unreachable criteria and use of a profiler to handle the more complex cases. Profiling in general can not *prove* anything about the unreachable of a piece of code, but may be a good heuristic for finding potentially unreachable code. Once a suspect piece of code is found, other methods, such as a more powerful code analysis tool, or even analysis by hand, could be used to decide whether the code is truly unreachable.

References

- [1] Debray, S. K., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (Mar. 2000), 378–415. (<http://doi.acm.org/10.1145/349214.349233>)
- Appel, A. W. 1998 *Modern Compiler Implementation in Java*. Cambridge University Press.
 - Muchnick S. S. 1997 *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Redundant code

Redundant code is a computer programming term for code, which may be source code or compiled code in a computer program, that has any form of redundancy, such as recomputing a value that has previously been calculated^[1] and is still available, code that is never executed (often called unreachable code), or code which is executed but has no external effect (e.g., does not change the output produced by a program), usually known as dead code.

A NOP instruction might be considered to be redundant code that has been explicitly inserted to pad out the instruction stream or introduce a time delay. Identifiers that are declared, but never referenced, are usually termed as redundant declarations. However, in some cases, a NOP can be used to create timing loops by "wasting time".

Example

```
int f (int x)
{
    int y=x*x;
    return x*x;
}
```

The second `x*x` expression is redundant code and can be replaced by a reference to the variable `y`. Alternatively, the definition `int y=x*x;` can instead be removed.

References

- [1] Debray, S. K., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (Mar. 2000), 378–415. (<http://doi.acm.org/10.1145/349214.349233>)

Jump threading

In computing, **jump threading** is a compiler optimization. In this pass, conditional jumps in the code that branch to identical or inverse tests are detected, and can be "threaded" through a second conditional test. This is easily done in a single pass through the program, following acyclic chained jumps until the compiler arrives at a fixed point.

Superoptimization

Superoptimization is the task of finding the optimal code sequence for a single, loop-free sequence of instructions. While garden-variety compiler optimizations really just *improve* code (real-world compilers generally cannot produce genuinely *optimal* code), a superoptimizer's goal is to find the optimal sequence at the outset.

The term superoptimization was first coined by Henry Massalin in his 1987 paper ^[1] and then later developed for integration within the GNU Compiler Collection (GSO ^[2] 1992). Recent work has further developed and extended this idea: (2001 ^[3], 2006 ^[4], 2006 ^[5]).

Typically, superoptimization is performed via exhaustive search in the space of valid instruction sequences. While this is an expensive technique, and therefore impractical for general-purpose compilers, it has been shown to be useful in optimizing performance-critical inner loops. Recent work ^[5] has used superoptimization to automatically generate general-purpose peephole optimizers.

External links

Publicly available superoptimizers

- GNU Superoptimizer (GSO) ^[6] (1992)
- TOAST Superoptimiser ^[7] (2006)
- The Aha! (A Hacker's Assistant) Superoptimizer ^[8] (and paper about it ^[9]) (2006)
- Stanford's Superoptimizer ^[10] (2006)
- PIC Microcontroller SuperOptimizer ^[11] (2007)

References

- [1] <http://portal.acm.org/citation.cfm?id=36194>
- [2] <http://portal.acm.org/citation.cfm?id=143146>
- [3] <http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-171.html>
- [4] http://dx.doi.org/10.1007/11799573_21
- [5] <http://theory.stanford.edu/~sbansal/pubs/asplos06.pdf>
- [6] <http://ftp.gnu.org/gnu/superopt>
- [7] <http://krr.cs.bath.ac.uk/index.php/TOAST>
- [8] <http://www.hackersdelight.org/aha.zip>
- [9] <http://www.hackersdelight.org/aha/aha.pdf>
- [10] <http://theory.stanford.edu/~sbansal/superoptimizer.html>
- [11] <http://freshmeat.net/projects/picsuperoptimizer/>

Loop optimization

In compiler theory, **loop optimization** plays an important role in improving cache performance, making effective use of parallel processing capabilities, and reducing overheads associated with executing loops. Most execution time of a scientific program is spent on loops. Thus a lot of compiler analysis and compiler optimization techniques have been developed to make the execution of loops faster.

Representation of computation and transformations

Since instructions inside loops can be executed repeatedly, it is frequently not possible to give a bound on the number of instruction executions that will be impacted by a loop optimization. This presents challenges when reasoning about the correctness and benefits of a loop optimization, specifically the representations of the computation being optimized and the optimization(s) being performed^[1].

Optimization via a sequence of loop transformations

Loop optimization can be viewed as the application of a sequence of specific **loop transformations** (listed below or in ^[2]) to the source code or Intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). Evaluating the benefit of a transformation or sequence of transformations can be quite difficult within this approach, as the application of one beneficial transformation may require the prior use of one or more other transformations that, by themselves, would result in reduced performance.

Common loop transformations

- fission/distribution : Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.
- fusion/combining : Another technique which attempts to reduce loop overhead. When two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data.
- interchange/permutation : These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.
- inversion : This technique changes a standard *while* loop into a *do/while* (a.k.a. *repeat/until*) loop wrapped in an *if* conditional, reducing the number of jumps by two, for cases when the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the *if* guard can be skipped.
- loop-invariant code motion : If a quantity is computed inside a loop during every iteration, and its value is the same for each iteration, it can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins. This is particularly important with the address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with loop inversion, because not all code is safe to be hoisted outside the loop.
- parallelization : a special case for Automatic parallelization focusing on loops, restructuring them to run efficiently on multiprocessor systems. It can be done automatically by compilers (named automatic parallelization) or manually (inserting parallel directives like OpenMP).
- reversal : Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations. Also, certain

architectures utilize looping constructs at Assembly language level that count in a single direction only (e.g. decrement-jump-if-not-zero (DJNZ)).

- scheduling
- skewing : Loop skewing takes a nested loop iterating over a multidimensional array, where each iteration of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop.
- Software pipelining : a type of out-of-order execution of loop iterations to hide the latencies of processor function units.
- splitting/peeling : Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. A useful special case is *loop peeling*, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.
- tiling/blocking : Loop tiling reorganizes a loop to iterate over blocks of data sized to fit in the cache.
- vectorization
- unrolling:Duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which may degrade performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead (except multiple instruction fetches & increased program load time), but requires that the number of iterations be known at compile time (except in the case of JIT compilers). Care must also be taken to ensure that multiple re-calculation of indexed variables is not a greater overhead than advancing pointers within the original loop.
- unswitching : Unswitching moves a conditional inside a loop outside of it by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional.

Other loop optimizations

- sectioning : First introduced for vectorizers, *loop-sectioning* (also known as *strip-mining*) is a loop-transformation technique for enabling SIMD-encodings of loops and improving memory performance. This technique involves each vector operation being done for a size less than or equal to the maximum vector length on a given vector machine. [3] [4]

The unimodular transformation framework

The **unimodular transformation** approach^[5] uses a single unimodular matrix to describe the combined result of a sequence of many of the above transformations. Central to this approach is the view of the set of all executions of a statement within n loops as a set of integer points in an n -dimensional space, with the points being executed in lexicographical order. For example, the executions of a statement nested inside an outer loop with index i and an inner loop with index j can be associated with the pairs of integers (i, j) . The application of a unimodular transformation corresponds to the multiplication of the points within this space by the matrix. For example, the interchange of two loops corresponds to the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.

A unimodular transformation is legal if it preserves the temporal sequence of all dependencies; measuring the performance impact of a unimodular transformation is more difficult. Imperfectly nested loops and some transformations (such as tiling) do not fit easily into this framework.

The polyhedral or constraint-based framework

The polyhedral model^[6] handles a wider class of programs and transformations than the unimodular framework. The set of executions of a set of statements within a possibly-imperfectly nested set of loops is seen as the union of a set of polytopes representing the executions of the statements. Affine transformations are applied to these polytopes, producing a description of a new execution order. The boundaries of the polytopes, the data dependencies, and the transformations are often described using systems of constraints, and this approach is often referred to as a **constraint-based** approach to loop optimization. For example, a single statement within an outer loop 'for i := 0 to n' and an inner loop 'for j := 0 to i+2' is executed once for each (i, j) pair such that $0 \leq i \leq n$ and $0 \leq j \leq i+2$.

Once again, a transformation is legal if it preserves the temporal sequence of all dependencies. Estimating the benefits of a transformation, or finding the best transformation for a given code on a given computer, remain the subject of ongoing research as of the time of this writing (2010).

References

- [1] Jean-Francois Collard, *Reasoning About Program Transformations*, 2003 Springer-Verlag. Discusses in depth the general question of representing executions of programs rather than program text in the context of static optimization.
- [2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. *Compiler transformations for high-performance computing*. Report No. UCB/CSD 93/781, Computer Science Division-EECS, University of California, Berkeley, Berkeley, California 94720, November 1993 (available at CiteSeer (<http://citeseer.ist.psu.edu/bacon93compiler.html>)). Introduces compiler analysis such as data dependence analysis and interprocedural analysis, as well as a very complete list of loop transformations
- [3] <http://software.intel.com/en-us/articles/strip-mining-to-optimize-memory-use-on-32-bit-intel-architecture/>
- [4] <http://download.oracle.com/docs/cd/E19205-01/819-5262/aeugr/index.html>
- [5] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, 1997 Morgan-Kauffman. Section 20.4.2 discusses loop optimization.
- [6] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

Induction variable

In computer science, an **induction variable** is a variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable.

For example, in the following loop, i and j are induction variables:

```
for (i=0; i < 10; ++i) {
    j = 17 * i;
}
```

Application to strength reduction

A common compiler optimization is to recognize the existence of induction variables and replace them with simpler computations; for example, the code above could be rewritten by the compiler as follows, on the assumption that the addition of a constant will be cheaper than a multiplication.

```
j = -17;
for (i = 0; i < 10; ++i) {
    j = j + 17;
}
```

This optimization is a special case of strength reduction.

Application to reduce register pressure

In some cases, it is possible to reverse this optimization in order to remove an induction variable from the code entirely. For example:

```
extern int sum;
int foo(int n) {
    int i, j;
    j = 5;
    for (i=0; i < n; ++i) {
        j += 2;
        sum += j;
    }
    return sum;
}
```

This function's loop has two induction variables: `i` and `j`. Either one can be rewritten as a linear function of the other; therefore, the compiler may optimize this code as if it had been written

```
extern int sum;
int foo(int n) {
    int i;
    for (i=0; i < n; ++i) {
        sum += (5 + 2*(i+1));
    }
    return sum;
}
```

Induction variable substitution

Induction variable substitution is a compiler transformation to recognize variables which can be expressed as functions of the indices of enclosing loops and replace them with induction variables with the expressions involving loop indices.

This transformation makes the relationship between the variables and loop indices explicit, which helps other compiler analysis, such as dependence analysis.

Example.

Input code:

```
int c, i;
c = 10;
for (i=0; i<10; i++)
{
    c = c + 5; // c is incremented by 5 for each loop iteration
}
```

Output code

```
int c, i;
c = 10;
for (i=0; i<10; i++)
{
```

```
c = 10 + 5*i; // c is explicitly expressed as a function of loop index
{
```

Non-linear induction variables

The same optimizations can be applied to induction variables that are not necessarily linear functions of the loop counter; for example, the loop

```
j = 1;
for (i=0; i < 10; ++i) {
    j = j << 1;
}
```

may be converted to

```
for (i=0; i < 10; ++i) {
    j = 1 << i+1;
}
```

References

- Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D. (1986), *Compilers: Principles, Techniques, and Tools* (2nd ed.), ISBN 978-0201100884
- Allen, Francis E.; Cocke, John; Kennedy, Ken (1981), "Reduction of Operator Strength", in Munchnik, Steven S.; Jones, Neil D., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, ISBN 978-0137296811
- Cocke, John; Kennedy, Ken (November 1977), "An algorithm for reduction of operator strength", *Communications of the ACM* **20** (11)
- Cooper, Keith; Simpson, Taylor; Vick, Christopher (1995), *Operator Strength Reduction* (<http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR95635-S.pdf>), Rice University, retrieved April 22, 2010

Loop fission

Loop fission (or **loop distribution**) is a compiler optimization technique attempting to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. The goal is to break down large loop body into smaller ones to achieve better utilization of locality of reference. It is the reverse action to loop fusion. This optimization is most efficient in multi-core processors that can split a task into multiple tasks for each processor.

Original loop	After loop fission
<pre>int i, a[100], b[100]; for (i = 0; i < 100; i++) { a[i] = 1; b[i] = 2; }</pre>	<pre>int i, a[100], b[100]; for (i = 0; i < 100; i++) { a[i] = 1; } for (i = 0; i < 100; i++) { b[i] = 2; }</pre>

Further reading

Kennedy, Ken; & Allen, Randy. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann. ISBN 1-55860-286-0.

Reference

Loop fission ^[1]

References

[1] <http://sc.tamu.edu/help/power/powerlearn/html/ScalarOptnw/sld031.htm>

Loop fusion

Loop fusion, also called **loop jamming**, is a compiler optimization, a loop transformation, which replaces multiple loops with a single one.

Example in C

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
    a[i] = 1;
for (i = 0; i < 100; i++)
    b[i] = 2;
```

is equivalent to:

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
    b[i] = 2;
}
```

Note

Some optimizations like this don't always improve the run-time performance. This is due to architectures that provide better performance if there are two loops rather than one, for example due to increased data locality within each loop. In those cases, a single loop may be transformed into two, which is called loop fission.

External links

- NullStone Loop Fusion ^[1]

References

[1] <http://www.nullstone.com/htmls/category/fusion.htm>

Loop inversion

Loop inversion is a compiler optimization, a loop transformation, which replaces a while loop by an if block containing a do..while loop.

Example in C

```
int i, a[100];
i = 0;
while (i < 100) {
    a[i] = 0;
    i++;
}
```

is equivalent to:

```
int i, a[100];
i = 0;
if (i < 100) {
    do {
        a[i] = 0;
        i++;
    } while (i < 100);
}
```

At a first glance, this seems like a bad idea: there's more code so it probably takes longer to execute. However, most modern CPUs use a pipeline for executing instructions. By nature, any jump in the code causes a pipeline stall. Let's watch what happens in Assembly-like Three address code version of the above code:

Example in Three-address code

```
i := 0
L1: if i >= 100 goto L2
    a[i] := 0
    i := i + 1
    goto L1
L2:
```

If i had been initialized at 100, the instructions executed at runtime would have been:

```
1: if i >= 100
2: goto L2
```

Let us assume that i had been initialized to some value less than 100. Now let us look at the instructions executed at the moment after i has been incremented to 99 in the loop:

```
1: goto L1
2: if i < 100
3: a[i] := 0
4: i := i + 1
5: goto L1
```

```
6: if i >= 100
7: goto L2
8: <<at L2>>
```

Now, let's look at the optimized version:

```
i := 0
if i >= 100 goto L2
L1: a[i] := 0
i := i + 1
if i < 100 goto L1
L2:
```

Again, let's look at the instructions executed if i is initialized to 100:

```
1: if i >= 100
2: goto L2
```

We didn't waste any cycles compared to the original version. Now consider the case where i has been incremented to 99:

```
1: if i < 100
2: goto L1
3: a[i] := 0
4: i := i + 1
5: if i < 100
6: <<at L2>>
```

As you can see, two *gotos* (and thus, two pipeline stalls) have been eliminated in the execution.

Additionally, Loop Inversion allows safe loop-invariant code motion to be performed.

Loop interchange

In compiler theory, **loop interchange** is the process of exchanging the order of two iteration variables.

For example, in the code fragment:

```
for i from 0 to 10
  for j from 0 to 20
    a[i,j] = i + j
```

loop interchange would result in:

```
for j from 0 to 20
  for i from 0 to 10
    a[i,j] = i + j
```

On occasion, such a transformation may lead to opportunities to further optimize, such as vectorization of the array assignments.

The utility of loop interchange

One major purpose of loop interchange is to improve the cache performance for accessing array elements. Cache misses occur if the contiguously accessed array elements within the loop come from a different cache line. Loop interchange can help prevent this. The effectiveness of loop interchange depends on and must be considered in light of the cache model used by the underlying hardware and the array model used by the compiler.

In the C programming language, the array elements from the same row are stored consecutively (Ex: a[1,1],a[1,2], a[1,3]...), namely **row-major**. On the other hand, FORTRAN programs store array elements from the same column together(Ex: a[1,1],a[2,1],a[3,1]...) , called **column-major**. Thus the order of two iteration variables in the first example is suitable for a C program while the second example is better for FORTRAN.^[1] Optimizing compilers can detect the improper ordering by programmers and interchange the order to achieve better cache performance.

Caveat

Like any other compiler optimization, loop interchange may lead to worse performance because cache performance is only part of the story. Take the following example:

```
do i = 1, 10000
  do j = 1, 1000
    a(i) = a(i) + b(j,i) * c(i)
  end do
end do
```

Loop interchange on this example can improve the cache performance of accessing b(j,i), but it will ruin the reuse of a(i) and c(i) in the inner loop, as it introduces two extra loads (for a(i) and for c(i)) and one extra store (for a(i)) during each iteration. As a result, the overall performance may be degraded after loop interchange.

Safety

It is not always safe to exchange the iteration variables due to dependencies between statements for the order in which they must execute. To determine whether a compiler can safely interchange loops, dependence analysis is required.

Further reading

Kennedy, Ken; & Allen, Randy. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann. ISBN 1-55860-286-0.

References

- [1] "Loop interchange" (<http://docs.hp.com/en/B3909-90003/ch05s08.html>). *Parallel Programming Guide for HP-UX Systems*. HP.
2010-03-05. .

Loop-invariant code motion

In computer programming, **loop-invariant code** consists of statements or expressions (in an imperative programming language) which can be moved outside the body of a loop without affecting the semantics of the program. **Loop-invariant code motion** (also called **hoisting** or **scalar promotion**) is a compiler optimization which performs this movement automatically.

Example

If we consider the following code sample, two optimizations can be easily applied.

```
for (int i = 0; i < n; i++) {  
    x = y + z;  
    a[i] = 6 * i + x * x;  
}
```

The calculations $x = y + z$ and $x * x$ can be moved outside the loop since within they are loop invariant—they do not change over the iterations of the loop—so the optimized code will be something like this:

```
x = y + z;  
t1 = x * x;  
for (int i = 0; i < n; i++) {  
    a[i] = 6 * i + t1;  
}
```

This code can be optimized further. For example, strength reduction could remove the two multiplications inside the loop ($6*i$ and $a[i]$), and induction variable elimination could then elide i completely. Since $6 * i$ must be in lock step with i itself, there is no need to have both.

Invariant code detection

Usually a reaching definitions analysis is used to detect whether a statement or expression is loop invariant.

For example, if all reaching definitions for the operands of some simple expression are outside of the loop, the expression can be moved out of the loop.

Benefits

Loop-invariant code which has been hoisted out of a loop is executed less often, providing a speedup. Another effect of this transformation is allowing constants to be stored in registers and not having to calculate the address and access the memory (or cache line) at each iteration.

However, if too many variables are created, there will be high register pressure, especially on processors with few registers, like the 32-bit x86. If the compiler runs out of registers, some variables will be spilled. To counteract this, the “opposite” optimization can be performed, rematerialization.

Further Reading

- Aho, Alfred V.; Sethi, Ravi; & Ullman, Jeffrey D. (1986). Compilers: Principles, Techniques, and Tools. Addison Wesley. ISBN 0-201-10088-6.

Loop nest optimization

Loop nest optimization (LNO) applies a set of loop transformations for the purpose of locality optimization or parallelization or other loop overhead reduction of the loop nests. One classical usage is to reduce memory access latency or the cache bandwidth necessary due to cache reuse for some common linear algebra algorithms.

Example: Matrix multiply

Many large mathematical operations on computers end up spending much of their time doing matrix multiplication. The operation is:

```
C = A×B
```

where A, B, and C are N×N arrays. Subscripts, for the following description, are in the form C [row] [column].

The basic loop is:

```
for (i = 0; i < N; i++)
{
    for (j=0; j < N; j++)
    {
        C[i][j] = 0;
        for (k = 0; k < N; k++)
            C[i][j] += A[k][j] * B[i][k];
    }
}
```

There are three problems to solve:

- Floating point additions take some number of cycles to complete. In order to keep an adder with multiple cycle latency busy, the code must update multiple accumulators in parallel.

- Machines can typically do just one memory operation per multiply-add, so values loaded must be reused at least twice.
- Typical PC memory systems can only sustain one 8-byte doubleword per 10–30 double-precision multiply-adds, so values loaded into the cache must be reused many times.

The original loop calculates the result for one entry in the result matrix at a time. By calculating a small block of entries simultaneously, the following loop reuses each loaded value twice, so that the inner loop has four loads and four multiply-adds, thus solving problem #2. By carrying four accumulators simultaneously, this code can keep a single floating point adder with a latency of 4 busy nearly all the time (problem #1). However, the code does not address the third problem. (Nor does it address the cleanup work necessary when N is odd. Such details will be left out of the following discussion.)

```
for (i = 0; i < N; i += 2)
{
    for (j = 0; j < N; j += 2)
    {
        acc00 = acc01 = acc10 = acc11 = 0;
        for (k = 0; k < N; k++)
        {
            acc00 += A[k][j + 0] * B[i + 0][k];
            acc01 += A[k][j + 1] * B[i + 0][k];
            acc10 += A[k][j + 0] * B[i + 1][k];
            acc11 += A[k][j + 1] * B[i + 1][k];
        }
        C[i + 0][j + 0] = acc00;
        C[i + 0][j + 1] = acc01;
        C[i + 1][j + 0] = acc10;
        C[i + 1][j + 1] = acc11;
    }
}
```

This code has had both the `i` and `j` iterations blocked by a factor of two, and had both the resulting two-iteration inner loops completely unrolled.

This code would run quite acceptably on a Cray Y-MP (built in the early 1980s), which can sustain 0.8 multiply-adds per memory operation to main memory. A machine like a 2.8 GHz Pentium 4, built in 2003, has slightly less memory bandwidth and vastly better floating point, so that it can sustain 16.5 multiply-adds per memory operation. As a result, the code above will run slower on the 2.8 GHz Pentium 4 than on the 166 MHz Y-MP!

A machine with a longer floating-point add latency or with multiple adders would require more accumulators to run in parallel. It is easy to change the loop above to compute a 3x3 block instead of a 2x2 block, but the resulting code is not always faster. The loop requires registers to hold both the accumulators and the loaded and reused A and B values. A 2x2 block requires 7 registers. A 3x3 block requires 13, which will not work on a machine with just 8 floating point registers in the ISA. If the CPU does not have enough registers, the compiler will schedule extra loads and stores to spill the registers into stack slots, which will make the loop run slower than a smaller blocked loop.

Matrix multiplication is like many other codes in that it can be limited by memory bandwidth, and that more registers can help the compiler and programmer reduce the need for memory bandwidth. This *register pressure* is why vendors of RISC CPUs, who intended to build machines more parallel than the general purpose x86 and 68000 CPUs, adopted 32-entry floating-point register files.

The code above does not use the cache very well. During the calculation of a horizontal stripe of C results, one horizontal stripe of B is loaded and the entire matrix A is loaded. For the entire calculation, C is stored once (that's good), B is loaded into the cache once (assuming a stripe of B fits in the cache with a stripe of A), but A is loaded N/ib times, where ib is the size of the strip in the C matrix, for a total of N^3/ib doubleword loads from main memory. In the code above, ib is 2.

The next step to reducing the memory traffic is to make ib as large as possible. We want it to be larger than the "balance" number reported by streams. In the case of one particular 2.8 GHz Pentium-4 system used for this example, the balance number is 16.5. The second code example above can't be extended directly, since that would require many more accumulator registers. Instead, we *block* the loop over i. (Technically, this is actually the second time we've blocked i, as the first time was the factor of 2.)

```
for (ii = 0; ii < N; ii += ib)
{
    for (j = 0; j < N; j += 2)
    {
        for (i = ii; i < ii + ib; i += 2)
        {
            acc00 = acc01 = acc10 = acc11 = 0;
            for (k = 0; k < N; k++)
            {
                acc00 += A[k][j + 0] * B[i + 0][k];
                acc01 += A[k][j + 1] * B[i + 0][k];
                acc10 += A[k][j + 0] * B[i + 1][k];
                acc11 += A[k][j + 1] * B[i + 1][k];
            }
            C[i + 0][j + 0] = acc00;
            C[i + 0][j + 1] = acc01;
            C[i + 1][j + 0] = acc10;
            C[i + 1][j + 1] = acc11;
        }
    }
}
```

With this code, we can set ib to be anything we like, and the number of loads of the A matrix will be reduced by that factor. This freedom has a cost: we are now keeping a $Nxib$ slice of the B matrix in the cache. So long as that fits, this code will not be limited by the memory system.

So what size matrix fits? Our example system, a 2.8 GHz Pentium 4, has a 16KB primary data cache. With $ib=20$, the slice of the B matrix in this code will be larger than the primary cache when $N > 100$. For problems larger than that, we'll need another trick.

That trick is reducing the size of the stripe of the B matrix by blocking the k loop, so that the stripe is of size $ib \times kb$. Blocking the k loop means that the C array will be loaded and stored N/kb times, for a total of $2*N^3/kb$ memory transfers. A is still transferred N/ib times, for N^3/ib transfers. So long as

$$2*N/kb + N/ib < N/balance$$

the machine's memory system will keep up with the floating point unit and the code will run at maximum performance. The 16KB cache of the Pentium 4 is not quite big enough: we might choose $ib=24$ and $kb=64$, thus using 12KB of the cache -- we don't want to completely fill it, since the C and A arrays have to have some room to flow through. These numbers comes within 20% of the peak floating-point speed of the processor.

Here is the code with loop k blocked.

```

for (ii = 0; ii < N; ii += ib)
{
    for (kk = 0; kk < N; kk += kb)
    {
        for (j=0; j < N; j += 2)
        {
            for(i = ii; i < ii + ib; i += 2 )
            {
                if (kk == 0)
                    acc00 = acc01 = acc10 = acc11 = 0;
                else
                {
                    acc00 = C[i + 0][j + 0];
                    acc01 = C[i + 0][j + 1];
                    acc10 = C[i + 1][j + 0];
                    acc11 = C[i + 1][j + 1];
                }
                for (k = kk; k < kk + kb; k++)
                {
                    acc00 += A[k][j + 0] * B[i + 0][k];
                    acc01 += A[k][j + 1] * B[i + 0][k];
                    acc10 += A[k][j + 0] * B[i + 1][k];
                    acc11 += A[k][j + 1] * B[i + 1][k];
                }
                C[i + 0][j + 0] = acc00;
                C[i + 0][j + 1] = acc01;
                C[i + 1][j + 0] = acc10;
                C[i + 1][j + 1] = acc11;
            }
        }
    }
}
}

```

The above code examples do not show the details of dealing with values of N which are not multiples of the blocking factors. Compilers which do loop nest optimization emit code to clean up the edges of the computation. For example, most LNO compilers would probably split the kk == 0 iteration off from the rest of the kk iterations, in order to remove the if statement from the i loop. This is one of the values of such a compiler: while it is straightforward to code the simple cases of this optimization, keeping all the details correct as the code is replicated and transformed is an error-prone process.

The above loop will only achieve 80% of peak flops on the example system when blocked for the 16KB L1 cache size. It will do worse on systems with even more unbalanced memory systems. Fortunately, the Pentium 4 has 256KB (or more, depending on the model) high-bandwidth level-2 cache as well as the level-1 cache. We are presented with a choice:

- We can adjust the block sizes for the level-2 cache. This will stress the processor's ability to keep many instructions in flight simultaneously, and there is a good chance it will be unable to achieve full bandwidth from the level-2 cache.

- We can block the loops again, again for the level-2 cache sizes. With a total of three levels of blocking (for the register file, for the L1 cache, and for the L2 cache), the code will minimize the required bandwidth at each level of the memory hierarchy. Unfortunately, the extra levels of blocking will incur still more loop overhead, which for some problem sizes on some hardware may be more time consuming than any shortcomings in the hardware's ability to stream data from the L2 cache.

External links

- Streams benchmark results ^[1], showing the overall balance between floating point operations and memory operations for many different computers
- "CHiLL: Composable High-Level Loop Transformation Framework" ^[2]

References

- [1] <http://www.cs.virginia.edu/stream/standard/Balance.html>
[2] <http://www.chunchen.info/chill>

Manifest expression

A **manifest expression** is a programming language construct that a compiler can analyse to deduce which values it can take without having to execute the program. This information can enable compiler optimizations, in particular loop nest optimization, and parallelization through data dependency analysis. An expression is called manifest if it is computed only from outer loop counters and constants (a more formal definition is given below).

When all control flow for a loop or condition is regulated by manifest expressions, it is called a **manifest loop** resp. **condition**.

Most practical applications of manifest expressions also require the expression to be integral and affine (or stepwise affine) in its variables.

Definition

A **manifest expression** is a compile time computable function which depends only on

- compile-time constants,
- manifest variable references, and
- loop counters of loops surrounding the expression.

A **manifest variable reference** is itself defined as a variable reference with

- a single, unambiguous definition of its value,
- which is itself a manifest expression.

The *single, unambiguous definition* is particularly relevant in procedural languages, where pointer analysis and/or data flow analysis is required to find the expression that defines the variable value. If several defining expressions are possible (e.g. because the variable is assigned in a condition), the variable reference is not manifest.

References

Feautrier, Paul (February 1991). "Dataflow analysis of array and scalar references" [1]. *International Journal of Parallel Programming* (Springer Netherlands) **20** (1): 23–53. doi:10.1007/BF01407931. issn = 0885-7458 (Print), 1573-7640 (Online).

References

[1] <http://citeseer.ist.psu.edu/367760.html>

Polytope model

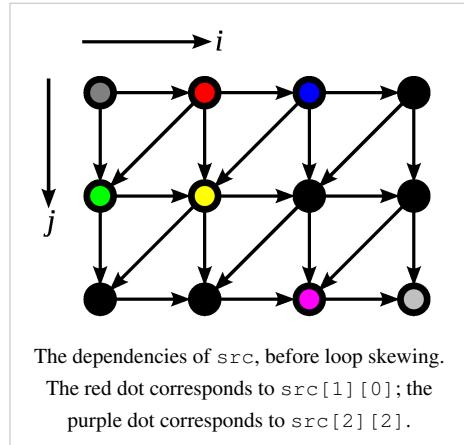
The **polyhedral model** (also called the **polytope method**) is a mathematical framework for loop nest optimization in program optimization. The polytope method treats each loop iteration within nested loops as lattice points inside mathematical objects called polytopes, performs affine transformations or more general non-affine transformations such as tiling on the polytopes, and then converts the transformed polytopes into equivalent, but optimized (depending on targeted optimization goal), loop nests through polyhedra scanning.

Detailed example

The following C code implements a form of error-distribution dithering similar to Floyd–Steinberg dithering, but modified for pedagogical reasons. The two-dimensional array `src` contains `h` rows of `w` pixels, each pixel having a grayscale value between 0 and 255 inclusive. After the routine has finished, the output array `dst` will contain only pixels with value 0 or value 255. During the computation, each pixel's dithering error is collected by adding it back into the `src` array. (Notice that `src` and `dst` are both read and written during the computation; `src` is not read-only, and `dst` is not write-only.)

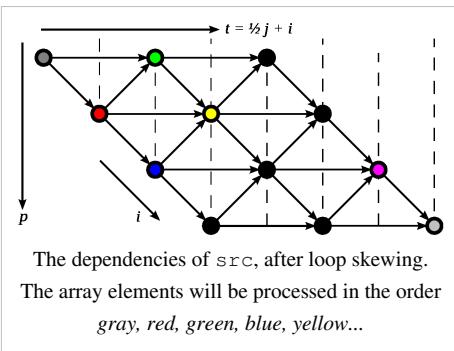
Each iteration of the inner loop modifies the values in `src[i][j]` based on the values of `src[i-1][j]`, `src[i][j-1]`, and `src[i+1][j-1]`. (The same dependencies apply to `dst[i][j]`.

For the purposes of loop skewing, we can think of `src[i][j]` and `dst[i][j]` as the same element.) We can illustrate the dependencies of `src[i][j]` graphically, as in the diagram on the right.



```
#define ERR(x,y) (dst[x][y] - src[x][y])

void dither(unsigned char **src, unsigned char **dst, int w, int h)
{
    int i, j;
    for (j = 0; j < h; ++j) {
        for (i = 0; i < w; ++i) {
            int v = src[i][j];
            if (i > 0)
                v -= ERR(i - 1, j) / 2;
            if (j > 0)
                v -= ERR(i, j - 1) / 4;
            if (j > 0 && i < w - 1)
                v -= ERR(i + 1, j - 1) / 4;
            dst[i][j] = (v < 128) ? 0 : 255;
            src[i][j] = (v < 0) ? 0 : (v < 255) ? v : 255;
        }
    }
}
```



Performing the affine transformation $(p, t) = (i, 2j + i)$ on the original dependency diagram gives us a new diagram, which is shown in the next image. We can then rewrite the code to loop on `p` and `t` instead of `i` and `j`, obtaining the following "skewed" routine.

```
void dither_skewed(unsigned char **src, unsigned char **dst, int w,
int h)
{
    int t, p;
    for (t = 0; t < (w + (2 * h)); ++t) {
        int pmin = max(t % 2, t - (2 * h) + 2);
        int pmax = min(t, w - 1);
        for (p = pmin; p <= pmax; p += 2) {
            int i = p;
            int j = (t - p) / 2;
            int v = src[i][j];
            if (i > 0)
                v -= ERR(i - 1, j) / 2;
            if (j > 0)
                v -= ERR(i, j - 1) / 4;
            if (j > 0 && i < w - 1)
                v -= ERR(i + 1, j - 1) / 4;
            dst[i][j] = (v < 128) ? 0 : 255;
            src[i][j] = (v < 0) ? 0 : (v < 255) ? v : 255;
        }
    }
}
```

External links and references

- "The basic polytope method" [1], tutorial by Martin Griebl containing diagrams of the pseudocode example above
- "Framework for polyhedral model"
- "Code Generation in the Polytope Model" [2] (1998). Martin Griebl, Christian Lengauer, and Sabine Wetzel
- "The CLoOG Polyhedral Code Generator" [3]
- "CodeGen+: Z-polyhedra scanning" [4]

References

[1] http://www.infosun.fmi.uni-passau.de/cl/loopo/doc/loopo_doc/node3.html

[2] <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.5675>

[3] <http://www.cloog.org/>

[4] <http://www.chunchen.info/omega>

Loop unwinding

Loop unwinding, also known as **loop unrolling**, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler.

The goal of loop unwinding is to increase a program's speed by reducing (or eliminating) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration;^[1] reducing branch penalties; as well as "hiding latencies, in particular, the delay in reading data from memory".^[2] Loops can be re-written instead as a repeated sequence of similar independent statements eliminating this overhead.^[3]

Advantages

The overhead in "tight" loops often consists of instructions to increment a pointer or index to the next element in an array (pointer arithmetic), as well as "end of loop" tests. If an optimizing compiler or assembler is able to pre-calculate offsets to each *individually referenced* array variable, these can be built into the machine code instructions directly, therefore requiring no additional arithmetic operations at run time (note that in the example given below this is not the case).

- Significant gains can be realized if the reduction in executed instructions more than compensates for any performance reduction caused by any increase in the size of the program.
- branch penalty is minimised.
- If the statements in the loop are independent of each other (i.e. where statements that occur earlier in the loop do not affect statements that follow them), the statements can potentially be executed in parallel.
- Can be implemented dynamically if the number of array elements is unknown at compile time

Optimizing compilers will sometimes perform the unrolling automatically or upon request.

Disadvantages

- The program code size increase after the unrolling, which is undesirable, particularly for embedded applications.
- Increased code size can also cause an increase in instruction cache misses, which may adversely affect performance.
- Unless performed transparently by an optimizing compiler, the code may become less readable.
- If the code in the body of the loop involves function calls, it may not be possible to combine unrolling with inlining, since the increase in code size might be excessive. Thus there can be a tradeoff between the two optimizations.
- Possible increased register usage in a single iteration to store temporary variables , which may reduce performance (though much will depend on possible optimizations).^[4]

Static/Manual Loop Unrolling

Manual (or static) loop unrolling involves the programmer analysing the loop and interpreting the iterations into a sequence of instructions which will reduce the loop overhead. This is in contrast to dynamic unrolling which is accomplished by the compiler.

A simple manual example in C Language

A procedure in a computer program is to delete 100 items from a collection. This is normally accomplished by means of a *for*-loop which calls the function *delete(item_number)*. If this part of the program is to be optimized, and the overhead of the loop requires significant resources compared to those for *delete(x)* loop, unwinding can be used to speed it up.

Normal loop	After loop unrolling
<pre>int x; for (x = 0; x < 100; x++) { delete(x); }</pre>	<pre>int x; for (x = 0; x < 100; x+=5) { delete(x); delete(x+1); delete(x+2); delete(x+3); delete(x+4); }</pre>

As a result of this modification, the new program has to make only 20 iterations, instead of 100. Afterwards, only 20% of the jumps and conditional branches need to be taken, and represents, over many iterations, a potentially significant decrease in the loop administration overhead. To produce the optimal benefit, no variables should be specified in the unrolled code that require pointer arithmetic. This usually requires "base plus offset" addressing, rather than indexed referencing.

On the other hand, this *manual* loop unrolling expands the source code size from 3 lines to 7, that have to be produced, checked, and debugged, and the compiler may have to allocate more registers to store variables in the expanded loop iteration . In addition, the loop control variables and number of operations inside the unrolled loop structure have to be chosen carefully so that the result is indeed the same as in the original code (assuming this is a later optimization on already working code). For example, consider the implications if the iteration count were not divisible by 5. The manual amendments required also become somewhat more complicated if the test conditions are variables. See also Duff's device.

Early complexity

In the simple case the loop control is merely an administrative overhead, that arranges the productive statements. The loop itself contributes nothing to the results desired, merely saving the programmer the tedium of replicating the code a hundred times which could have been done by a pre-processor generating the replications, or a text editor. Similarly, If-statements, and other flow control statements, could be replaced by code replication, except that code bloat can be the result. Computer programs easily track the combinations, but programmers find this repetition boring and make mistakes. Consider:

Normal loop	After loop unrolling
<pre>for i:=1:8 do if i mod 2 = 0 then do_evenstuff(i) else do_oodstuff(i); next i;</pre>	<pre>do_oodstuff(1); do_evenstuff(2); do_oodstuff(3); do_evenstuff(4); do_oodstuff(5); do_evenstuff(6); do_oodstuff(7); do_evenstuff(8);</pre>

But of course, the code performed need not be the invocation of a procedure, and this next example involves the index variable in computation:

Normal loop	After loop unrolling
<pre>x(1) = 1; For i:=2:9 do x(i):=x(i - 1)*i; print i,x(i); next i; </pre></pre>	<pre>x(1):=1; x(2):=x(1)*2; print 2,x(2); x(3):=x(2)*3; print 3,x(3); ...etc. .</pre>

which, if compiled, might produce a lot of code (*print* statements being notorious) but further optimization is possible. This example make reference only to $x(i)$ and $x(i - 1)$ in the loop (the latter only to develop the new value $x(i)$) therefore, given that there is no later reference to the array x developed here, its usages could be replaced by a simple variable. Such a change would however mean a simple variable *whose value is changed* whereas if staying with the array, the compiler's analysis might note that the array's values are constant, each derived from a previous constant, and therfore carry forwards the constant values so that the code becomes

```
print 2,2;
print 3,6;
print 4,24;
...etc.
```

It would be quite a surprise if the compiler were to recognise $x(n) = \text{Factorial}(n)$.

In general, the content of a loop might be large, involving intricate array indexing. These cases are probably best left to optimizing compilers to unroll. Replicating innermost loops might allow many possible optimisations yet yield only a small gain unless n is large.

Unrolling WHILE loops

A pseudocode WHILE loop - similar to the following -

Normal loop	After loop unrolling	Unrolled & "tweaked" loop
<pre>WHILE (condition) DO action ENDWHILE</pre>	<pre>WHILE (condition) DO action IF NOT(condition) THEN GOTO break action IF NOT(condition) THEN GOTO break action ENDWHILE LABEL break: .</pre>	<pre>IF (condition) THEN REPEAT { action IF NOT(condition) THEN GOTO break action IF NOT(condition) THEN GOTO break action } WHILE (condition) LABEL break:</pre>

Unrolling is faster because the ENDWHILE (that will be compiled to a jump to the start of the loop) will be executed 66% less often.

Even better, the "tweaked" pseudocode example, that may be performed automatically by some optimizing compilers, eliminating unconditional jumps altogether.

Dynamic unrolling

Since the benefits of loop unrolling are frequently dependent on the size of an array - that may often not be known until run time, JIT compilers (for example) can determine whether to invoke a "standard" loop sequence or instead generate a (relatively short) sequence of individual instructions for each element. This flexibility is one of the advantages of just-in-time techniques versus static or manual optimization in the context of loop unrolling. In this situation, it is often with relatively small values of 'n' where the savings are still useful - requiring quite small (if any) overall increase in program size (that might be included just once, as part of a standard library).

Assembly language programmers (including optimizing compiler writers) are also able to benefit from the technique of dynamic loop unrolling, using a method similar to that used for efficient branch tables. Here the advantage is greatest where the maximum offset of any referenced field in a particular array is less than the maximum offset that can be specified in a machine instruction (which will be flagged by the assembler if exceeded). The example below is for IBM/360 or Z/Architecture assemblers and assumes a field of 100 bytes (at offset zero) is to be copied from array 'FROM' to array 'TO' - both having element lengths of 256 bytes with 50 entries

Assembler example (IBM/360 or Z/Architecture)

For an x86 example, see External Links.

```
* initialize all the registers to point to arrays etc (R14 is return address)
        LM    R15,R2,INIT          load R15= '16', R0=number in array, R1--> 'FROM array', R2--> 'TO array'
LOOP    EQU    *
        SR    R15,R0              get 16 minus the number in the array
        BNP   ALL                if n > 16 need to do all of the sequence, then repeat
* (if # entries = zero, R15 will now still be 16, so all the MVC's will be bypassed)
* calculate an offset (from start of MVC sequence) for unconditional branch to 'unwound' MVC loop
        MH    R15,=AL2(ILEN)      multiply by length of (MVC..) instruction (=6 in this example)
        B    ALL(R15)             indexed branch instruction (to MVC with drop through)
* Assignment (move) table - (first entry has maximum allowable offset with single register = X'FOO' in this example)
ALL    MVC    15*256(100,R2),15*256(R1)    * move 100 bytes of 16th entry from array 1 to array 2 (with drop through)
ILEN   EQU    *-ALL            length of (MVC...) instruction sequence; in this case =6
```

```

MVC 14*256(100,R2),14*256(R1) *
MVC 13*256(100,R2),13*256(R1) *
MVC 12*256(100,R2),12*256(R1) * All 16 of these 'move character' instructions use base plus offset addressing
MVC 11*256(100,R2),11*256(R1) * and each to/from offset decreases by the length of one array element (256).
MVC 10*256(100,R2),10*256(R1) * This avoids pointer arithmetic being required for each element up to a
MVC 09*256(100,R2),09*256(R1) * maximum permissible offset within the instruction of X'FFF'. The instructions
MVC 08*256(100,R2),08*256(R1) * are in order of decreasing offset, so the first element in the set is moved
MVC 07*256(100,R2),07*256(R1) * last.
MVC 06*256(100,R2),06*256(R1) *
MVC 05*256(100,R2),05*256(R1) *
MVC 04*256(100,R2),04*256(R1) *
MVC 03*256(100,R2),03*256(R1) *
MVC 02*256(100,R2),02*256(R1) *
MVC 01*256(100,R2),01*256(R1) move 100 bytes of 2nd entry
MVC 00*256(100,R2),00*256(R1) move 100 bytes of 1st entry

*
S R0,MAXM1 reduce Count = remaining entries to process
BNPR R14 ... no more, so return to address in R14
AH R1,=AL2(16*256) increment 'FROM' register pointer beyond first set
AH R2,=AL2(16*256) increment 'TO' register pointer beyond first set
L R15,MAXM1 re-load (maximum MVC's) in R15 (destroyed by calculation earlier)
B LOOP go and execute loop again

*
* ----- Define static Constants and variables (These could be passed as parameters) -----
INIT DS 0A 4 addresses (pointers) to be pre-loaded with a 'LM' instruction
MAXM1 DC A(16) maximum MVC's
N DC A(50) number of actual entries in array (a variable, set elsewhere)
DC A(FROM) address of start of array 1 ("pointer")
DC A(TO) address of start of array 2 ("pointer")
* ----- Define static Arrays (These could be dynamically acquired) -----
FROM DS 50CL256 array of (max) 50 entries of 256 bytes each
TO DS 50CL256 array of (max) 50 entries of 256 bytes each

```

In this example, approximately 202 instructions would be required with a 'conventional' loop (50 iterations) whereas the above dynamic code would require only about 89 instructions (or a saving of approximately 56%). If the array had consisted of only 2 entries, it would still execute in approximately the same time as the original unwound loop. The increase in code size is only about 108 bytes - even if there are thousands of entries in the array. Similar techniques can of course be used where multiple instructions are involved, as long as the combined instruction length is adjusted accordingly. For example, in this same example, if it is required to clear the rest of each array entry to nulls immediately after the 100 byte field copied, an additional clear instruction, 'XC xx*256+100(156,R1),xx*256+100(R2)', can be added immediately after every MVC in the sequence (where 'xx' matches the value in the MVC above it).

It is, of course, perfectly possible to generate the above code 'inline' using a single assembler macro statement, specifying just 4 or 5 operands (or alternatively, make it into a library subroutine, accessed by a simple call, passing a list of parameters), making the optimization readily accessible to inexperienced programmers.

C example

The following example demonstrates dynamic loop unrolling for a simple program written in C. Unlike the assembler example above, pointer/index arithmetic is still generated by the compiler in this example because a variable (*i*) is still used to address the array element. Full optimization is only possible if absolute indexes are used in the replacement statements.

```
#include<stdio.h>

#define TOGETHER (8)

int main(void)
{
    int i = 0;
    int entries = 50;                                /* total number to
process      */
    int repeat;                                       /* number of times
for while.. */
    int left = 0;                                     /* remainder
(process later)   */

/* If the number of elements is not be divisible by BLOCKSIZE,
   */
/* get repeat times required to do most processing in the while loop
   */
repeat = (entries / TOGETHER);                      /* number of times
to repeat    */
left = (entries % TOGETHER);                         /* calculate
remainder      */

/* Unroll the loop in 'bunches' of 8
   */
while( repeat-- > 0 )
{
    printf("process(%d)\n", i      );
    printf("process(%d)\n", i + 1);
    printf("process(%d)\n", i + 2);
    printf("process(%d)\n", i + 3);
    printf("process(%d)\n", i + 4);
    printf("process(%d)\n", i + 5);
    printf("process(%d)\n", i + 6);
    printf("process(%d)\n", i + 7);

/* update the index by amount processed in one go
   */
    i += TOGETHER;
}
```

```

/* Use a switch statement to process remaining by jumping to the case
label      */
/* at the label that will then drop through to complete the set
   */
switch (left)
{
    case 7 : printf("process(%d)\n", i + 6);           /* process and rely
on drop through drop through */
    case 6 : printf("process(%d)\n", i + 5);
    case 5 : printf("process(%d)\n", i + 4);
    case 4 : printf("process(%d)\n", i + 3);
    case 3 : printf("process(%d)\n", i + 2);
    case 2 : printf("process(%d)\n", i + 1);           /* two left
   */
    case 1 : printf("process(%d)\n", i );               /* just one left to
process
   */
    case 0 : ;                                         /* none left
   */
}
}

```

Further reading

Kennedy, Ken; Allen, Randy (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann. ISBN 1-55860-286-0.

References

- [1] Ullman, Jeffrey D.; Aho, Alfred V. (1977). *Principles of compiler design*. Reading, Mass: Addison-Wesley Pub. Co. pp. 471–2.
ISBN 0-201-10073-8.
- [2] Petersen, W.P., Arbenz, P. (2004). *Introduction to Parallel Computing*. Oxford University Press. p. 10.
- [3] Nicolau, Alexandru (1985). *Loop Quantization: Unwinding for Fine-Grain Parallelism Exploitation*. Dept. of Computer Science Technical Report. Ithaca, NY: Cornell University. OCLC 14638257.
- [4] Sarkar, Vivek (2001). "Optimized Unrolling of Nested Loops" (<http://www.springerlink.com/content/g36002133451w774/>).
International Journal of Parallel Programming 29 (5): 545–581. doi:10.1023/A:1012246031671..

External links

- Chapter 7, pages 8 to 10 (<http://nondot.org/~sabre/Mirrored/GraphicsProgrammingBlackBook/gpbb7.pdf>), of Michael Abrash's *Graphics Programming Black Book* is about loop unrolling, with an example in x86 assembly.
- (<http://www.cs.uh.edu/~jhuang/JCH/JC/loop.pdf>) Generalized Loop Unrolling, gives a concise introduction.

Loop splitting

Loop splitting is a compiler optimization technique. It attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range.

Loop peeling

Loop peeling is a special case of loop splitting which splits any problematic first (or last) few iterations from the loop and performs them outside of the loop body.

Suppose a loop was written like this:

```
int p = 10;
for (int i=0; i<10; ++i)
{
    y[i] = x[i] + x[p];
    p = i;
}
```

Notice that $p = 10$ only for the first iteration, and for all other iterations, $p = i - 1$. A compiler can take advantage of this by unwinding (or "peeling") the first iteration from the loop.

After peeling the first iteration, the code would look like this:

```
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i)
{
    y[i] = x[i] + x[i-1];
}
```

This equivalent form eliminates the need for the variable p inside the loop body.

Loop peeling was introduced in gcc in version 3.4.

Brief history of the term

Apparently the term was for the first time used by Cannings, Thompson and Skolnick^[1] in their 1976 paper on computational models for (human) inheritance. There the term was used to denote a method for collapsing phenotypic information onto parents. From there the term was used again in their papers, including their seminal paper on probability functions on complex pedigrees.^[2]

In compiler technology, the term first turned up in late 1980s papers on VLIW and superscalar compilation, including^[3] and^[4]

Further reading

- Kennedy, Ken; & Allen, Randy. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann. ISBN 1-55860-286-0.
- [1] Cannings, C.; Thompson, EA; Skolnick, HH (1976). "The recursive derivation of likelihoods on complex pedigrees". *Advances in Applied Probability* **8** (4): 622–625.
- [2] Cannings, C.; Thompson, EA; Skolnick, HH (1978). "Probability functions on complex pedigrees". *Advances in Applied Probability* **10** (1): 26–61.
- [3] Callahan, D; Kennedy, K (1988). "Compiling Programs for Distributed-memory Multiprocessors". *The Journal of Supercomputing* **2** (2): 151–169.
- [4] Mahlke, SA; Lin, DC; Chen, WY; Hank, RE; Bringman, RA (1992). "Effective compiler support for predicated execution using the hyperblock". 25th Annual International Symposium on Microarchitecture. pp. 45–54.

Loop tiling

Loop tiling, also known as **loop blocking**, or **strip mine and interchange**, is a loop optimization used by compilers to make the execution of certain types of loops more efficient.

Overview

Loop tiling partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of loop iteration space leads to partitioning of large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements. An ordinary loop

```
for(i=0; i<N; ++i) {
    ...
}
```

can be blocked with a block size B by replacing it with

```
for(j=0; j<N; j+=B)
    for(i=j; i<min(N, j+B); ++i) {
        ...
    }
```

where min() is a function returning the minimum of its arguments.

Example

The following is an example of matrix vector multiplication. There are three arrays, each with 100 elements. The code does not partition the arrays into smaller sizes.

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++) {
    c[i] = 0;
    for (j = 0; j < n; j++) {
        c[i] = c[i] + a[i][j] * b[j];
    }
}
```

After we apply loop tiling using 2×2 blocks, our code looks like:

```

int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
    c[i] = 0;
    for (j = 0; j < n; j += 2) {
        for (x = i; x < min(i + 2, n); x++) {
            for (y = j; y < min(j + 2, n); y++) {
                c[x] = c[x] + a[x][y] * b[y];
            }
        }
    }
}

```

The original loop iteration space is n by n . The accessed chunk of array $a[i, j]$ is also n by n . When n is too large and the cache size of the machine is too small, the accessed array elements in one loop iteration (for example, $i = 1, j = 1$ to n) may cross cache lines, causing cache misses.

Another example using an algorithm for matrix multiplication:

Original matrix multiplication:

```

DO I = 1, M
DO K = 1, M
DO J = 1, M
    Z(J, I) = Z(J, I) + X(K, I) * Y(J, K)

```

After loop tiling B*B:

```

DO K2 = 1, M, B
DO J2 = 1, M, B
    DO I = 1, M
        DO K1 = K2, MIN(K2 + B - 1, M)
            DO J1 = J2, MIN(J2 + B - 1, M)
                Z(J1, I) = Z(J1, I) + X(K1, I) * Y(J1, K1)

```

It is not always easy to decide what value of tiling size is optimal for one loop because it demands an accurate estimate of accessed array regions in the loop and the cache size of the target machine. The order of loop nests (loop interchange) also plays an important role in achieving better cache performance.

Further reading

1. Wolfe, M. *More Iteration Space Tiling*. Supercomputing'89, pages 655—664, 1989.
2. Wolf, M. E. and Lam, M. *A Data Locality Optimizing Algorithm*. PLDI'91, pages 30—44, 1991.
3. Irigoin, F. and Triolet, R. *Supernode Partitioning*. POPL'88, pages 319—329, 1988.
4. Xue, J. *Loop Tiling for Parallelism*. Kluwer Academic Publishers. 2000.
5. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 63—74, April 1991.

Loop unswitching

Loop unswitching is a compiler optimization. It moves a conditional inside a loop outside of it by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional. This can improve the parallelization of the loop. Since modern processors can operate fast on vectors this increases the speed.

Here is a simple example. Suppose we want to add the two arrays x and y and also do something depending on the variable w . We have the following C code:

```
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++) {
    x[i] = x[i] + y[i];
    if (w)
        y[i] = 0;
}
```

The conditional inside this loop makes it difficult to safely parallelize this loop. When we unswitch the loop, this becomes:

```
int i, w, x[1000], y[1000];
if (w) {
    for (i = 0; i < 1000; i++) {
        x[i] = x[i] + y[i];
        y[i] = 0;
    }
} else {
    for (i = 0; i < 1000; i++) {
        x[i] = x[i] + y[i];
    }
}
```

While the loop unswitching may double the amount of code written, each of these new loops may now be separately optimized.

Loop unswitching was introduced in gcc in version 3.4.

Interprocedural optimization

Interprocedural optimization (IPO) is a collection of compiler techniques used in computer programming to improve performance in programs containing many frequently used functions of small or medium length. IPO differs from other compiler optimization because it analyzes the entire program; other optimizations look at only a single function, or even a single block of code.

IPO seeks to reduce or eliminate duplicate calculations, inefficient use of memory, and to simplify iterative sequences such as loops. If there is a call to another routine that occurs within a loop, IPO analysis may determine that it is best to inline that. Additionally, IPO may re-order the routines for better memory layout and locality.

IPO may also include typical compiler optimizations on a whole-program level, for example dead code elimination, which removes code that is never executed. To accomplish this, the compiler tests for branches that are never taken and removes the code in that branch. IPO also tries to ensure better use of constants. Modern compilers offer IPO as an option at compile-time. The actual IPO process may occur at any step between the human-readable source code and producing a finished executable binary program.

Analysis

The objective, as ever, is to have the program run as swiftly as possible; the problem, as ever, is that it is not possible for a compiler to analyse a program and always correctly determine what it will do, still less what the programmer may have intended for it to do. By contrast, human programmers start at the other end with a purpose, and attempt to produce a program that will achieve it, preferably without expending a lot of thought in the process. So, the hope is that an optimising compiler will aid us by bridging the gap.

For various reasons, including readability, programs are frequently broken up into a number of procedures, which handle a few general cases. However, the generality of each procedure may result in wasted effort in specific usages. Interprocedural optimization represents an attempt at reducing this waste.

Suppose you have a procedure that evaluates $F(x)$, and your code requests the result of $F(6)$ and then later, $F(6)$ again. Surely this second evaluation is unnecessary: the result could have been saved, and referred to later instead. This assumes that F is a pure function. Particularly, this simple optimization is foiled the moment that the implementation of $F(x)$ is impure; that is, its execution involves reference to parameters other than the explicit argument 6 that have been changed between the invocations, or side effects such as printing some message to a log, counting the number of evaluations, accumulating the CPU time consumed, preparing internal tables so that subsequent invocations for related parameters will be facilitated, and so forth. Losing these side effects via non-evaluation a second time may be acceptable, or may not: a design issue beyond the scope of compilers.

More generally, aside from optimization, the second reason to use procedures is to avoid duplication of code that would be the same, or almost the same, each time the actions performed by the procedure are desired. A general approach to optimization would therefore be to reverse this: some or all invocations of a certain procedure are replaced by the respective code, with the parameters appropriately substituted. The compiler will then try to optimize the result.

Example

```
Program example;
    integer b;                                %A variable "global" to the procedure Silly.
Procedure Silly(a,x)
    if x < 0 then a:=x + b else a:=-6;
End Silly;                                  %Reference to b, not a parameter, makes Silly "impure" in general.
    integer a,x;                            %These variables are visible to Silly only if parameters.
```

```

x:=7; b:=5;
Silly(a,x); Print x;
Silly(x,a); Print x;
Silly(b,b); print b;
End example;

```

If the parameters to *Silly* are passed by value, the actions of the procedure have no effect on the original variables, and since *Silly* does nothing to its environment (read from a file, write to a file, modify global variables such as *b*, etc.) its code plus all invocations may be optimized away entirely, leaving the value of *a* undefined (which doesn't matter) so that just the *print* statements remain, and they for constant values.

If instead the parameters are passed by reference, then action on them within *Silly* does indeed affect the originals. This is usually done by passing the machine address of the parameters to the procedure so that the procedure's adjustments are to the original storage area. Thus in the case of call by reference, procedure *Silly* has an effect. Suppose that its invocations are expanded in place, with parameters identified by address: the code amounts to

```

x:=7; b:=5;
if x < 0 then a:=x + b else a:=-6; print x;    %a is changed.
if a < 0 then x:=a + b else x:=-6; print x;    %Because the parameters are swapped.
if b < 0 then b:=b + b else b:=-6; print b;    %Two versions of variable b in Silly, plus the global usage.

```

The compiler could then in this rather small example follow the constants along the logic (such as it is) and find that the predicates of the if-statements are constant and so...

```

x:=7; b:=5;
a:=-6; print 7;                      %b is not referenced, so this usage remains "pure".
x:=-1; print -1;                     %b is referenced...
b:=-6; print -6;                     %b is modified via its parameter manifestation.

```

And since the assignments to *a*, *b* and *x* deliver nothing to the outside world - they do not appear in output statements, nor as input to subsequent calculations (whose results in turn *do* lead to output, else they also are needless) - there is no point in this code either, and so the result is

```

print 7;
print -1;
print -6;

```

A variant method for passing parameters that appears to be "by reference" is copy-in, copy-out whereby the procedure works on a local copy of the parameters whose values are copied back to the originals on exit from the procedure. If the procedure has access to the same parameter but in different ways as in invocations such as *Silly(a,a)* or *Silly(a,b)*, discrepancies can arise. So, if the parameters were passed by copy-in, copy-out in left-to-right order then *Silly(b,b)* would expand into

```

p1:=b; p2:=b;      %Copy in. Local variables p1 and p2 are equal.
if p2 < 0 then p1:=p2 + b else p1:=-6;      %Thus p1 may no longer equal p2.
b:=p1; b:=p2;      %Copy out. In left-to-right order, the value from p1 is overwritten.

```

And in this case, copying the value of *p1* (which has been changed) to *b* is pointless, because it is immediately overwritten by the value of *p2*, which value has not been modified within the procedure from its original value of *b*, and so the third statement becomes

```

print 5;          %Not -6

```

Such differences in behavior are likely to cause puzzlement, exacerbated by questions as to the order in which the parameters are copied: will it be left to right on exit as well as entry? These details are probably not carefully explained in the compiler manual, and if they are, they will likely be passed over as being not relevant to the immediate task and long forgotten by the time a problem arises. If (as is likely) temporary values are provided via a stack storage scheme, then it is likely that the copy-back process will be in the reverse order to the copy-in, which in this example would mean that $p1$ would be the last value returned to b instead.

Incidentally, when procedures modify their parameters, it is important to be sure that any constants supplied as parameters will not have their value changed (constants can be held in memory just as variables are) lest subsequent usages of that constant (made via reference to its memory location) go awry. This can be accomplished by compiler-generated code copying the constant's value into a temporary variable whose address is passed to the procedure, and if its value is modified, no matter; it is never copied back to the location of the constant. Put another way, a carefully-written test program can report on whether parameters are passed by value or reference, and if used, what sort of copy-in and copy-out scheme. However, variation is endless: simple parameters might be passed by copy whereas large aggregates such as arrays might be passed by reference; simple constants such as zero might be generated by special machine codes (such as Clear, or LoadZ) while more complex constants might be stored in memory tagged as read-only with any attempt at modifying it resulting in immediate program termination, etc.

In General

This example is extremely simple, although complications are already apparent. More likely it will be a case of many procedures, having a variety of deducible or programmer-declared properties that may enable the compiler's optimizations to find some advantage. Any parameter to a procedure might be read only, be written to, be both read and written to, or be ignored altogether giving rise to opportunities such as constants not needing protection via temporary variables, but what happens in any given invocation may well depend on a complex web of considerations. Other procedures, especially function-like procedures will have certain behaviours that in specific invocations may enable some work to be avoided: for instance, the Gamma function, if invoked with an integer parameter, could be converted to a calculation involving integer factorials.

Some computer languages enable (or even require) assertions as to the usage of parameters, and might further offer the opportunity to declare that variables have their values restricted to some set (for instance, $6 < x \leq 28$) thus providing further grist for the optimisation process to grind through, and also providing worthwhile checks on the coherence of the source code to detect blunders. But this is never enough - only some variables can be given simple constraints, while others would require complex specifications: how might it be specified that variable P is to be a prime number, and if so, is or is not the value 1 included? Complications are immediate: what are the valid ranges for a day-of-month D given that M is a month number? And are all violations worthy of immediate termination? Even if all that could be handled, what benefit might follow? And at what cost? Full specifications would amount to a re-statement of the program's function in another form and quite aside from the time the compiler would consume in processing them, they would thus be subject to bugs. Instead, only simple specifications are allowed with run-time range checking provided.

In cases where a program reads no input (as in the example), one could imagine the compiler's analysis being carried forward so that the result will be no more than a series of print statements, or possibly some loops expediently generating such values. Would it then recognise a program to generate prime numbers, and convert to the best-known method for doing so, or, present instead a reference to a library? Unlikely! In general, arbitrarily complex considerations arise (the Entscheidungsproblem) to preclude this, and there is no option but to run the code with limited improvements only.

History

For procedural, or Algol-like languages, interprocedural analysis and optimization appears to have entered commercial practice in the early 1970s. IBM's PL/I Optimizing Compiler performed interprocedural analysis to understand the side effects of both procedure calls and exceptions (cast, in PL/I terms as "on conditions")^[1] and in papers by Fran Allen.^[2] ^[3] Work on compilation of APL was, of necessity, interprocedural.^[4] ^[5]

The techniques of interprocedural analysis and optimization were the subject of academic research in the 1980s and 1990s. They re-emerged into the commercial compiler world in the early 1990s with compilers from both Convex (the "Application Compiler" for the Convex C4) and from Ardent (the compiler for the Ardent Titan). These compilers demonstrated that the technologies could be made sufficiently fast to be acceptable in a commercial compiler; subsequently interprocedural techniques have appeared in a number of commercial and non-commercial systems.

Flags and implementation

The Intel C/C++ compilers allow whole-program IPO. The flag to enable interprocedural optimizations for a single file is -ip, the flag to enable interprocedural optimization across all files in the program is -ipo^[6] ^[7].

The GNU GCC compiler has function inlining, which is turned on by default at -O3, and can be turned on manually via passing the switch (-finline-functions) at compile time ^[8]. GCC version 4.1 has a new infrastructure for inter-procedural optimization ^[9].

Also Gcc has options for IPO: -fwhole-program -combine.

The Microsoft C Compiler, integrated into Visual Studio, also supports interprocedural optimization. ^[10]

The Clang supports IPO at optimization level -O4.

References

- [1] Thomas C. Spillman, "Exposing side effects in a PL/I optimizing compiler", in *Proceedings of IFIPS 1971*, North-Holland Publishing Company, pages 376-381.
- [2] Frances E. Allen, "Interprocedural Data Flow Analysis", IFIPS Proceedings, 1974.
- [3] Frances E. Allen, and Jack Schwartz, "Determining the Data Flow Relationships in a Collection of Procedures", IBM Research Report RC 4989, Aug. 1974.
- [4] Philip Abrams, "An APL Machine", Stanford University Computer Science Department, Report STAN-CS-70-158, February, 1970.
- [5] Terrence C. Miller, "Tentative Compilation: A Design for an APL Compiler", Ph.D. Thesis, Yale University, 1978.
- [6] "Intel compiler 8 documentation" (http://www.tacc.utexas.edu/services/userguides/intel8/cc/c_ug/lin1149.htm). .
- [7] Intel Visual Fortran Compiler 9.1, Standard and Professional Editions, for Windows* - Intel Software Network ([http://www.intel.com/cd/software/products/asmo-na/eng/compilers/fwin/278834.htm#Interprocedural_Optimization_\(IPO\)](http://www.intel.com/cd/software/products/asmo-na/eng/compilers/fwin/278834.htm#Interprocedural_Optimization_(IPO)))
- [8] "GCC optimization options" (<http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html#Optimize-Options>). .
- [9] "GCC interprocedural optimizations" (http://gcc.gnu.org/wiki/Interprocedural_optimizations). .
- [10] "Visual Studio Optimization" (http://msdn.microsoft.com/vstudio/tour/vs2005_guided_tour/VS2005pro/Framework/CPlusAdvancedProgramOptimization.htm). .

External Links

- How to trick C/C++ compilers into generating terrible code? (<http://www.futurechips.org/tips-for-power-coders/how-to-trick-cc-compilers-into-generating-terrible-code.html>)

Whole program optimization

In computer programming, **whole program optimization** is the compiler optimization of a program using information about all the modules in the program. Normally, optimizations are performed on a per module (per function), "compiland", basis; but this approach, while easier to write and test and less demanding of resources during the compilation itself, do not allow certainty about the safety of a number of optimizations such as aggressive inlining and thus cannot perform them even if they would actually turn out to be efficiency gains that do not change the semantics of the emitted object code.

Adaptive optimization

Adaptive optimization is a technique in computer science that performs dynamic recompilation of portions of a program based on the current execution profile. With a simple implementation, an adaptive optimizer may simply make a trade-off between Just-in-time compilation and interpreting instructions. At another level, adaptive optimization may take advantage of local data conditions to optimize away branches and to use inline expansion to decrease context switching.

Consider a hypothetical banking application that handles transactions one after another. These transactions may be checks, deposits, and a large number of more obscure transactions. When the program executes, the actual data may consist of clearing tens of thousands of checks without processing a single deposit and without processing a single check with a fraudulent account number. An adaptive optimizer would compile assembly code to optimize for this common case. If the system then started processing tens of thousands of deposits instead, the adaptive optimizer would recompile the assembly code to optimize the new common case. This optimization may include inlining code or moving error processing code to secondary cache.

Deoptimization

In some systems, notably the Java Virtual Machine, execution over a range of bytecode instructions can be provably reversed. This allows an adaptive optimizer to make risky assumptions about the code. In the above example, the optimizer may assume all transactions are checks and all account numbers are valid. When these assumptions prove incorrect, the adaptive optimizer can 'unwind' to a valid state and then interpret the byte code instructions correctly.

External links

- CiteSeer for "Adaptive Optimization in the Jalapeño JVM (2000)" ^[1] by Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney. Contains links to the full paper in various formats.
- HP's Dynamo ^[2]. Interesting code morphing system.

References

- [1] <http://citeseer.ist.psu.edu/arnold00adaptive.html>
[2] <http://arstechnica.com/reviews/1q00/dynamo/dynamo-1.html>

Lazy evaluation

In programming language theory, **lazy evaluation** or **call-by-need**^[1] is an evaluation strategy which delays the evaluation of an expression until its value is actually required (non-strict evaluation) and also avoids repeated evaluations (sharing).^[2] ^[3] The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name.

The benefits of lazy evaluation include: performance increases due to avoiding unnecessary calculations, avoiding error conditions in the evaluation of compound expressions, the capability of constructing potentially infinite data structures, and the capability of defining control structures as abstractions instead of as primitives. Lazy evaluation can lead to reduction in memory footprint, since values are created when needed.^[4] However, with lazy evaluation, it is difficult to combine with imperative features such as exception handling and input/output, because the order of operations becomes indeterminate. Also, debugging is difficult.^[5]

The opposite of lazy actions is eager evaluation, sometimes known as strict evaluation. Eager evaluation is the evaluation behavior used in most programming languages.

History

Lazy evaluation was introduced for the lambda calculus by (Wadsworth 1971) and for programming languages independently by (Henderson & Morris 1976) and (Friedman & Wise 1976).^[6]

Applications

Delayed evaluation is used particularly in functional languages. When using delayed evaluation, an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value. That is, a statement such as `x := expression;` (i.e. the assignment of the result of an expression to a variable) clearly calls for the expression to be evaluated and the result placed in `x`, but what actually is in `x` is irrelevant until there is a need for its value via a reference to `x` in some later expression whose evaluation could itself be deferred, though eventually the rapidly-growing tree of dependencies would be pruned in order to produce some symbol rather than another for the outside world to see.^[7]

Some programming languages delay evaluation of expressions by default, and some others provide functions or special syntax to delay evaluation. In Miranda and Haskell, evaluation of function arguments is delayed by default. In many other languages, evaluation can be delayed by explicitly suspending the computation using special syntax (as with Scheme's "delay" and "force" and OCaml's "lazy" and "Lazy.force") or, more generally, by wrapping the expression in a thunk (delayed computation). The object representing such an explicitly delayed evaluation is called a future or promise. Perl 6 uses lazy evaluation of lists, so one can assign infinite lists to variables and use them as arguments to functions, but unlike Haskell and Miranda, Perl 6 doesn't use lazy evaluation of arithmetic operators and functions by default.^[7]

Delayed evaluation has the advantage of being able to create calculable infinite lists without infinite loops or size matters interfering in computation. For example, one could create a function that creates an infinite list (often called a *stream*) of Fibonacci numbers. The calculation of the *n*-th Fibonacci number would be merely the extraction of that element from the infinite list, forcing the evaluation of only the first *n* members of the list.^[8] ^[9]

For example, in Haskell, the list of all Fibonacci numbers can be written as:^[9]

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

In Haskell syntax, ":" prepends an element to a list, `tail` returns a list without its first element, and `zipWith` uses a specified function (in this case addition) to combine corresponding elements of two lists to produce a third.^[8]

Provided the programmer is careful, only the values that are required to produce a particular result are evaluated. However, certain calculations may result in the program attempting to evaluate an infinite number of elements; for example, requesting the length of the list or trying to sum the elements of the list with a fold operation would result in the program either failing to terminate or running out of memory.

Control structures

Even in most eager languages *if* statements evaluate in a lazy fashion.

```
if a then b else c
```

evaluates (a), then if and only if (a) evaluates to true does it evaluate (b), otherwise it evaluates (c). That is, either (b) or (c) will not be evaluated. Conversely, in an eager language the expected behavior is that

```
define f(x,y) = 2*x  
set k = f(e,5)
```

will still evaluate (e) and (f) when computing (k). However, user-defined control structures depend on exact syntax, so for example

```
define g(a,b,c) = if a then b else c  
l = g(h,i,j)
```

(i) and (j) would both be evaluated in an eager language. While in

```
l' = if h then i else j
```

(i) or (j) would be evaluated, but never both.

Lazy evaluation allows control structures to be defined normally, and not as primitives or compile-time techniques. If (i) or (j) have side effects or introduce run time errors, the subtle differences between (l) and (l') can be complex. As most programming languages are Turing-complete, it is possible to introduce user-defined lazy control structures in eager languages as functions, though they may depart from the language's syntax for eager evaluation: Often the involved code bodies (like (i) and (j)) need to be wrapped in a function value, so that they are executed only when called.

Short-circuit evaluation of Boolean control structures is sometimes called "lazy".

Other uses

In computer windowing systems, the painting of information to the screen is driven by "expose events" which drive the display code at the last possible moment. By doing this, they avoid the computation of unnecessary display content.^[10]

Another example of laziness in modern computer systems is copy-on-write page allocation or demand paging, where memory is allocated only when a value stored in that memory is changed.^[10]

Laziness can be useful for high performance scenarios. An example is the Unix mmap functionality. mmap provides "demand driven" loading of pages from disk, so that only those pages actually touched are loaded into memory, and unnecessary memory is not allocated.

Controlling eagerness in lazy languages

In lazy programming languages such as Haskell, although the default is to evaluate expressions only when they are demanded, it is possible in some cases to make code more eager—or conversely, to make it more lazy again after it has been made more eager. This can be done by explicitly coding something which forces evaluation (which may make the code more eager) or avoiding such code (which may make the code more lazy). *Strict* evaluation usually implies eagerness, but they are technically different concepts.

However, there is an optimisation implemented in some compilers called strictness analysis, which, in some cases, allows the compiler to infer that a value will always be used. In such cases, this may render the programmer's choice of whether to force that particular value or not, irrelevant, because strictness analysis will force strict evaluation.

In Haskell, marking constructor fields strict means that their values will always be demanded immediately. The `seq` function can also be used to demand a value immediately and then pass it on, which is useful if a constructor field should generally be lazy. However, neither of these techniques implements *recursive* strictness—for that, a function called `deepSeq` was invented.

Also, pattern matching in Haskell 98 is strict by default, so the `~` qualifier has to be used to make it lazy.

Notes

- [1] Hudak 1989, p. 384
- [2] David Anthony Watt; William Findlay (2004). *Programming language design concepts* (<http://books.google.com/books?id=vogP3P2L4tgC&pg=PA367>). John Wiley and Sons. pp. 367–368. ISBN 9780470853207. . Retrieved 30 December 2010.
- [3] Reynolds 1998, p. 307
- [4] Chris Smith (22 October 2009). *Programming F#* (<http://books.google.com/books?id=gzVdyw2WoXMC&pg=PA79>). O'Reilly Media, Inc.. p. 79. ISBN 9780596153649. . Retrieved 31 December 2010.
- [5] OCaml and lazy evaluation (<http://caml.inria.fr/pub/ml-archives/caml-list/1999/10/7b5cd5d7e12a9876637ada0ee3ed3e06.en.html>)
- [6] Reynolds 1998, p. 312
- [7] Philip Wadler (2006). *Functional and logic programming: 8th international symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006 : proceedings* (<http://books.google.com/books?id=gZzLFFZfc1sC&pg=PA149>). Springer. p. 149. ISBN 9783540334385. . Retrieved 14 January 2011.
- [8] Daniel Le Métayer (2002). *Programming languages and systems: 11th European Symposium on Programming, ESOP 2002, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002 : proceedings* (<http://books.google.com/books?id=dYZyzp-I9hQC&pg=PA129>). Springer. pp. 129–132. ISBN 9783540433637. . Retrieved 14 January 2011.
- [9] Association for Computing Machinery; ACM Special Interest Group on Programming Languages (1 January 2002). *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell '02) : Pittsburgh, Pennsylvania, USA ; October 3, 2002* (<http://books.google.com/books?id=hsBQAAAAMAAJ>). Association for Computing Machinery. p. 40. ISBN 9781581136050. . Retrieved 14 January 2011.
- [10] Lazy and Speculative Execution (<http://research.microsoft.com/en-us/um/people/blampson/slides/lazyandspeculative.ppt>) Butler Lampson Microsoft Research OPODIS, Bordeaux, France 12 December 2006

References

- Hudak, Paul (September 1989). "Conception, Evolution, and Application of Functional Programming Languages" (<http://portal.acm.org/citation.cfm?id=72554>). *ACM Computing Surveys* **21** (3): 383–385.
- Reynolds, John C. (1998). *Theories of programming languages* (<http://books.google.com/books?id=HkI01IHJMcQC&pg=PA307>). Cambridge University Press. ISBN 978052159414.

Further reading

- Wadsworth, Christopher P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University
- Henderson, Peter; Morris, James H. (January 1976). "A Lazy Evaluator" (<http://portal.acm.org/citation.cfm?id=811543>). *Conference Record of the Third ACM symposium on Principles of Programming Languages*.

- Friedman, D. P.; Wise, David S. (1976). S. Michaelson and R. Milner, ed. "Cons should not evaluate its arguments" (<http://www.cs.indiana.edu/pub/techreports/TR44.pdf>). *Automata Languages and Programming Third International Colloquium* (Edinburgh University Press).

Design patterns

- John Hughes. "Why functional programming matters" (<http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>). *The Computer Journal* - Special issue on lazy functional programming (<http://comjnl.oxfordjournals.org/content/32/2.toc>). Volume 32 Issue 2, April 1989.
- Philip Wadler. "How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages" (<http://www.springerlink.com/content/y7450255v2670167/>). *Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, 1985, Volume 201/1985, 113-128.

Blog posts by computer scientists

- Robert Harper. "The Point of Laziness" (<http://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/>)
- Lennart Augustsson. "More points for lazy evaluation" (<http://augustss.blogspot.com/2011/05/more-points-for-lazy-evaluation-in.html>)

External links

- Lazy Evaluation (<http://c2.com/cgi/wiki?LazyEvaluation>) at the Portland Pattern Repository
- Lazy evaluation (http://haskell.org/haskellwiki/Haskell/Lazy_evaluation) at Haskell Wiki
- Functional programming in Python becomes lazy (http://gnosis.cx/publish/programming/charming_python_b13.html)
- Lazy function argument evaluation (<http://www.digitalmars.com/d/lazy-evaluation.html>) in the D programming language
- Lazy evaluation macros (http://nemerle.org/Lazy_evaluation) in Nemerle
- Lazy programming and lazy evaluation (<http://www-128.ibm.com/developerworks/linux/library/l-lazyprog.html>) in Scheme
- Lambda calculus in Boost Libraries (http://spirit.sourceforge.net/dl_docs/phoenix-2/libs/spirit/phoenix/doc/html/phoenix/introduction.html) in C++ programming language
- Lazy Evaluation (<http://perldesignpatterns.com/?LazyEvaluation>) in Perl

Partial evaluation

Not to be confused with partial application.

In computing, **partial evaluation** is a technique for several different types of program optimization by specialization. The most straightforward application is to produce new programs which run faster than the originals while being guaranteed to behave in the same way. More advanced uses include compiling by partially evaluating an interpreter with the program to be compiled as its input; generating compilers by partially evaluating a partial evaluator with an interpreter for the source language concerned as its input; and finally generating a compiler-generator by partially evaluating a partial evaluator with itself as its input.

A computer program, *prog*, is seen as a mapping of input data into output data:

$$\textit{prog} : I_{\text{static}} \times I_{\text{dynamic}} \rightarrow O$$

I_{static} , the *static data*, is the part of the input data known at compile time.

The partial evaluator transforms $\langle \textit{prog}, I_{\text{static}} \rangle$ into $\textit{prog}^* : I_{\text{dynamic}} \rightarrow O$ by precomputing all static input at compile time. \textit{prog}^* is called the "residual program" and should run more efficiently than the original program. The act of partial evaluation is said to "residualize" *prog* to \textit{prog}^* .

Futamura projections

A particularly interesting example of this, first described in the 1970s by Yoshihiko Futamura,^[1] is when *prog* is an interpreter for a programming language.

If I_{static} is source code designed to run inside said interpreter, then partial evaluation of the interpreter with respect to this data/program produces \textit{prog}^* , a version of the interpreter that only runs that source code, is written in the implementation language of the interpreter, does not require the source code to be resupplied, and runs faster than the original combination of the interpreter and the source. In this case \textit{prog}^* is effectively a compiled version of I_{static} .

This technique is known as the first Futamura projection, of which there are three:

1. Specializing an interpreter for given source code, yielding an executable
2. Specializing the specializer for the interpreter (as applied in #1), yielding a compiler
3. Specializing the specializer for itself (as applied in #2), yielding a tool that can convert any interpreter to an equivalent compiler

References

- [1] Yoshihiko Futamura's Website (<http://fi.ftmr.info/>)
- Yoshihiko Futamura (1971). "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler" (<http://www.brics.dk/~hosc/local/HOSC-12-4-pp381-391.pdf>). *Systems, Computers, Controls* 2 (5): 45–50. Reprinted in *Higher-Order and Symbolic Computation* 12 (4): 381–391, 1999, with a foreword.
- Charles Consel and Olivier Danvy (1993). "Tutorial Notes on Partial Evaluation". *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*: 493–501.

External links

- Neil D. Jones, Carsten K. Gomard, and Peter Sestoft: *Partial Evaluation and Automatic Program Generation* (1993) (<http://www.itu.dk/people/sestoft/pebook/>) Book, full text available online.
- partial-eval.org (<http://partial-eval.org>) - a large "Online Bibliography of Partial Evaluation Research".
- 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99) (<http://www.brics.dk/~pepm99/>)
- C++ Templates as Partial Evaluation, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99) (<http://osl.iu.edu/~tveldhui/papers/pepm99/>)
- C++ Templates as Partial Evaluation (<http://arxiv.org/pdf/cs.PL/9810010>) a different version including Catat (pdf)
- Applying Dynamic Partial Evaluation to dynamic, reflective programming languages (<http://people.csail.mit.edu/gregs/dynamic-pe.html>)

Profile-guided optimization

Profile-guided optimization (PGO, sometimes pronounced as “pogo”^[1]) is a compiler optimization technique in computer programming to improve program runtime performance. In contrast to traditional optimization techniques that solely use the source code, PGO uses the results of test runs of the instrumented program to optimize the final generated code.^[2] The compiler is used to access data from a sample run of the program across a representative input set. The data indicates which areas of the program are executed more frequently, and which areas are executed less frequently. All optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions. The caveat, however, is that the sample of data fed to the program during the profiling stage must be statistically representative of the typical usage scenarios; otherwise, profile-guided feedback has the potential of harming the overall performance of the final build instead of improving it.

Implementations

Examples of compilers that implement PGO are the Intel C++ Compiler and Fortran compilers, GNU Compiler Collection compilers, Sun Studio, and the Microsoft Visual C++ Compiler.

References

- [1] "Microsoft Visual C++ Team Blog" (<http://blogs.msdn.com/vcblog/archive/2008/11/12/pogo.aspx>). .
- [2] "Intel Fortran Compiler 10.1, Professional and Standard Editions, for Mac OS X" (<http://www.intel.com/cd/software/products/asmo-na/eng/compilers/fmac/267426.htm#pgo>). .

Automatic parallelization

Automatic parallelization, also **auto parallelization**, **autoparallelization**, or **parallelization**, the last one of which implies automation when used in context, refers to converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine. The goal of automatic parallelization is to relieve programmers from the tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation.^[1]

The programming control structures on which autoparallelization places the most focus are loops, because, in general, most of the execution time of a program takes place inside some form of loop. A parallelizing compiler tries to split up a loop so that its iterations can be executed on separate processors concurrently.

Compiler parallelization analysis

The *compiler* usually conducts two passes of analysis before actual parallelization in order to determine the following:

- Is it safe to parallelize the loop? Answering this question needs accurate dependence analysis and alias analysis
- Is it worthwhile to parallelize it? This answer requires a reliable estimation (modeling) of the program workload and the capacity of the parallel system.

The first pass of the compiler performs a data dependence analysis of the loop to determine whether each iteration of the loop can be executed independently of the others. Data dependence can sometimes be dealt with, but it may incur additional overhead in the form of message passing, synchronization of shared memory, or some other method of processor communication.

The second pass attempts to justify the parallelization effort by comparing the theoretical execution time of the code after parallelization to the code's sequential execution time. Somewhat counterintuitively, code does not always benefit from parallel execution. The extra overhead that can be associated with using multiple processors can eat into the potential speedup of parallelized code.

Example

The Fortran code below can be auto-parallelized by a compiler because each iteration is independent of the others, and the final result of array *z* will be correct regardless of the execution order of the other iterations.

```
do i=1, n
    z(i) = x(i) + y(i)
enddo
```

On the other hand, the following code cannot be auto-parallelized, because the value of *z*(*i*) depends on the result of the previous iteration, *z*(*i*-1).

```
do i=2, n
    z(i) = z(i-1)*2
enddo
```

This does not mean that the code cannot be parallelized. Indeed, it is equivalent to

```
do i=2, n
    z(i) = z(1)*2** (i-1)
```

```
enddo
```

However, current parallelizing compilers are not usually capable of bringing out these parallelisms automatically, and it is questionable whether this code would benefit from parallelization in the first place.

Difficulties

Automatic parallelization by compilers or tools is very difficult due to the following reasons:

- dependence analysis is hard for code using indirect addressing, pointers, recursion, and indirect function calls;
- loops have an unknown number of iterations;
- accesses to global resources are difficult to coordinate in terms of memory allocation, I/O, and shared variables.

Workaround

Due to the inherent difficulties in full automatic parallelization, several easier approaches exist to get a parallel program in higher quality. They are:

- Allow programmers to add "hints" to their programs to guide compiler parallelization, such as HPF for distributed memory systems and OpenMP or OpenHMPP for shared memory systems.
- Build an interactive system between programmers and parallelizing tools/compilers. Notable examples are Vector Fabrics' vfAnalyst, SUIF Explorer (The Stanford University Intermediate Format compiler), the Polaris compiler, and ParaWise (formally CAPTools).
- Hardware-supported speculative multithreading.

Historical parallelizing compilers

Most research compilers for automatic parallelization consider Fortran programs, because Fortran makes stronger guarantees about aliasing than languages such as C. Typical examples are:

- Rice Fortran D compiler
- Vienna Fortran compiler
- Paradigm compiler
- Polaris compiler
- SUIF compiler

References

- [1] Fox, Geoffrey; Roy Williams, Giuseppe Messina (1994). *Parallel Computing Works!*. Morgan Kaufmann. pp. 575,593.
ISBN 978-1558602533.

Loop scheduling

In parallel computing, **loop scheduling** is the problem of assigning proper iterations of parallelizable loops among n processors to achieve load balancing and maintain data locality with minimum dispatch overhead.

Typical loop scheduling methods are:

- static even scheduling: evenly divide loop iteration space into n chunks and assign each chunk to a processor
- dynamic scheduling: a chunk of loop iteration is dispatched at runtime by an idle processor. When the chunk size is 1 iteration, it is also called self-scheduling.
- guided scheduling: similar to dynamic scheduling, but the chunk sizes per dispatch keep shrinking until reaching a preset value.

Vectorization

Vectorization, in parallel computing, is a special case of parallelization, in which software programs that by default perform one operation at a time on a single thread are modified to perform multiple operations simultaneously.

Vectorization is the more limited process of converting a computer program from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation which processes one operation on multiple pairs of operands at once. The term comes from the convention of putting operands into vectors or arrays.

Vector processing is a major feature of both conventional computers and modern supercomputers.

Automatic vectorization is major research topic in computer science; seeking methods that would allow a compiler to convert scalar programs into vectorized programs without human assistance.

Background

Early computers generally had one logic unit that sequentially executed one instruction on one operand pair at a time. Computer programs and programming languages were accordingly designed to execute sequentially. Modern computers can do many things at once. Many optimizing compilers feature auto-vectorization, a compiler feature where particular parts of sequential programs are transformed into equivalent parallel ones, to produce code which will well utilize a vector processor. For a compiler to produce such efficient code for a programming language intended for use on a vector-processor would be much simpler, but, as much real-world code is sequential, the optimization is of great utility.

Loop vectorization converts procedural loops that iterate over multiple pairs of data items and assigns a separate processing unit to each pair. Most programs spend most of their execution times within such loops. Vectorizing loops can lead to orders of magnitude performance gains without programmer intervention, especially on large data sets. Vectorization can sometimes instead slow execution because of pipeline synchronization, data movement timing and other issues.

Intel's MMX, SSE and Power Architecture's AltiVec and ARM's NEON instruction sets support such vectorized loops.

Many constraints prevent or hinder vectorization. Loop dependence analysis identifies loops that can be vectorized, relying on the data dependence of the instructions inside loops.

Guarantees

Automatic vectorization, like any loop optimization or other compile-time optimization, must exactly preserve program behavior.

Data dependencies

All dependencies must be respected during execution to prevent incorrect outcomes.

In general, loop invariant dependencies and lexically forward dependencies can be easily vectorized, and lexically backward dependencies can be transformed into lexically forward. But these transformations must be done safely, in order to assure the dependence between **all statements** remain true to the original.

Cyclic dependencies must be processed independently of the vectorized instructions.

Data precision

Integer precision (bit-size) must be kept during vector instruction execution. The correct vector instruction must be chosen based on the size and behavior of the internal integers. Also, with mixed integer types, extra care must be taken to promote/demote them correctly without losing precision. Special care must be taken with sign extension (because multiple integers are packed inside the same register) and during shift operations, or operations with carry bits that would otherwise be taken into account.

Floating-point precision must be kept as well, unless IEEE-754 compliance is turned off, in which case operations will be faster but the results may vary slightly. Big variations, even ignoring IEEE-754 usually means programmer error. The programmer can also force constants and loop variables to single precision (default is normally double) to execute twice as many operations per instruction.

Theory

To vectorize a program, the compiler's optimizer must first understand the dependencies between statements and re-align them, if necessary. Once the dependencies are mapped, the optimizer must properly arrange the implementing instructions changing appropriate candidates to vector instructions, which operate on multiple data items.

Building the dependency graph

The first step is to build the dependency graph, identifying which statements depend on which other statements. This involves examining each statement and identifying every data item that the statement accesses, mapping array access modifiers to functions and checking every access' dependency to all others in all statements. Alias analysis can be used to certify that the different variables access (or intersects) the same region in memory.

The dependency graph contains all local dependencies with distance not greater than the vector size. So, if the vector register is 128 bits, and the array type is 32 bits, the vector size is $128/32 = 4$. All other non-cyclic dependencies should not invalidate vectorization, since there won't be any concurrent access in the same vector instruction.

Suppose the vector size is the same as 4 ints:

```
for (i = 0; i < 128; i++) {  
    a[i] = a[i+16]; // 16 > 4, safe to ignore  
    a[i] = a[i+1]; // 1 < 4, stays on dependency graph  
}
```

Clustering

Using the graph, the optimizer can then cluster the strongly connected components (SCC) and separate vectorizable statements from the rest.

For example, consider a program fragment containing three statement groups inside a loop: (SCC1+SCC2), SCC3 and SCC4, in that order, in which only the second group (SCC3) can be vectorized. The final program will then contain three loops, one for each group, with only the middle one vectorized. The optimizer cannot join the first with the last without violating statement execution order, which would invalidate the necessary guarantees.

Detecting idioms

Some non-obvious dependencies can be further optimized based on specific idioms.

For instance, the following self-data-dependencies can be vectorized because the value of the right-hand values (RHS) are fetched and then stored on the left-hand value, so there is no way the data will change within the assignment.

```
a[i] = a[i] + a[i+1];
```

Self-dependence by scalars can be vectorized by variable elimination.

General framework

The general framework for loop vectorization is split into four stages:

- **Prelude:** Where the loop-independent variables are prepared to be used inside the loop. This normally involves moving them to vector registers with specific patterns that will be used in vector instructions. This is also the place to insert the run-time dependence check. If the check decides vectorization is not possible, branch to **Cleanup**.
- **Loop(s):** All vectorizes (or not) loops, separated by SCCs clusters in order of appearance in the original code.
- **Postlude:** Return all loop-independent variables, inductions and reductions.
- **Cleanup:** Implement plain (non-vectorized) loops for iterations at the end of a loop that are not a multiple of the vector size) or for when run-time checks prohibit vector processing.

Run-time vs. compile-time

Some vectorizations cannot be fully checked at compile time. Compile-time optimization requires an explicit array index. Library functions can also defeat optimization if the data they process is supplied by the caller. Even in these cases, run-time optimization can still vectorize loops on-the-fly.

This run-time check is made in the **prelude** stage and directs the flow to vectorized instructions if possible, otherwise reverting to standard processing, depending on the variables that are being passed on the registers or scalar variables.

The following code can easily be vectorized on compile time, as it doesn't have any dependence on external parameters. Also, the language guarantees that neither will occupy the same region in memory as any other variable, as they are local variables and live only in the execution stack.

```
int a[128];
int b[128];
// initialize b

for (i = 0; i<128; i++)
    a[i] = b[i] + 5;
```

On the other hand, the code below has no information on memory positions, because the references are pointers and the memory they point to lives in the heap.

```
int *a = malloc(128*sizeof(int));
int *b = malloc(128*sizeof(int));
// initialize b

for (i = 0; i<128; i++, a++, b++)
    *a = *b + 5;
// ...
// ...
// ...
free(b);
free(a);
```

A quick run-time check on the address of both *a* and *b*, plus the loop iteration space (128) is enough to tell if the arrays overlap or not, thus revealing any dependencies.

Techniques

An example would be a program to multiply two vectors of numeric data. A scalar approach would be something like:

```
for (i = 0; i < 1024; i++)
    C[i] = A[i]*B[i];
```

This could be vectorized to look something like:

```
for (i = 0; i < 1024; i+=4)
    C[i:i+3] = A[i:i+3]*B[i:i+3];
```

Here, *C[i:i+3]* represents the four array elements from *C[i]* to *C[i+3]* and the vector processor can perform four operations for a single vector instruction. Since the four vector operations complete in roughly the same time like one scalar instruction, the vector approach can run up to four times faster than the original code.

There are two distinct compiler approaches: one based on conventional vectorization technique and the other based on loop unrolling.

Loop-level automatic vectorization

This technique, used for conventional vector machines, tries to find and exploit SIMD parallelism at the loop level. It consists of two major steps as follows.

1. Find an innermost loop that can be vectorized
2. Transform the loop and generate vector codes

In the first step, the compiler looks for obstacles that can prevent vectorization. A major obstacle for vectorization is true data dependency shorter than the vector length. Other obstacles include function calls and short iteration counts.

Once the loop is determined to be vectorizable, the loop is stripmined by the vector length and each scalar instruction within the loop body is replaced with the corresponding vector instruction. Below, the component transformations for this step are shown using the above example.

- After stripmining

```
for (i = 0; i < 1024; i+=4)
    for (ii = 0; ii < 4; ii++)
        C[i+ii] = A[i+ii]*B[i+ii];
```

- After loop distribution using temporary arrays

```
for (i = 0; i < 1024; i+=4)
{
    for (ii = 0; ii < 4; ii++) tA[ii] = A[i+ii];
    for (ii = 0; ii < 4; ii++) tB[ii] = B[i+ii];
    for (ii = 0; ii < 4; ii++) tC[ii] = tA[ii]*tB[ii];
    for (ii = 0; ii < 4; ii++) C[i+ii] = tC[ii];
}
```

- After replacing with vector codes

```
for (i = 0; i < 1024; i+=4)
{
    vA = vec_ld( &A[i] );
    vB = vec_ld( &B[i] );
    vC = vec_mul( vA, vB );
    vec_st( vC, &C[i] );
}
```

Basic block level automatic vectorization

This relatively new technique specifically targets modern SIMD architectures with short vector lengths^[1]. Although loops can be unrolled to increase the amount of SIMD parallelism in basic blocks, this technique exploits SIMD parallelism within basic blocks rather than loops. The two major steps are as follows.

1. The innermost loop is unrolled by a factor of the vector length to form a large loop body.
2. Isomorphic scalar instructions (that perform the same operation) are packed into a vector instruction if dependencies do not prevent doing so.

To show step-by-step transformations for this approach, the same example is used again.

- After loop unrolling (by the vector length, assumed to be 4 in this case)

```
for (i = 0; i < 1024; i+=4)
{
    sA0 = ld( &A[i+0] );
    sB0 = ld( &B[i+0] );
    sC0 = sA0 * sB0;
    st( sC0, &C[i+0] );
    ...
    sA3 = ld( &A[i+3] );
    sB3 = ld( &B[i+3] );
    sC3 = sA3 * sB3;
    st( sC3, &C[i+3] );
}
```

- After packing

```
for (i = 0; i < 1024; i+=4)
{
    (sA0,sA1,sA2,sA3) = ld( &A[i+0:i+3] );
    (sB0,sB1,sB2,sB3) = ld( &B[i+0:i+3] );
    (sC0,sC1,sC2,sC3) = (sA0,sA1,sA2,sA3) * (sB0,sB1,sB2,sB3);
    st( (sC0,sC1,sC2,sC3), &C[i+0:i+3] );
}
```

- After code generation

```
for (i = 0; i < 1024; i+=4)
{
    vA = vec_ld( &A[i] );
    vB = vec_ld( &B[i] );
    vC = vec_mul( vA, vB );
    vec_st( vC, &C[i] );
}
```

Here, sA1, sB1, ... represent scalar variables and vA, vB, and vC represent vector variables.

Most automatically vectorizing commercial compilers use the conventional loop-level approach except the IBM XL Compiler^[1], which uses both.

In the presence of control flow

The presence of if-statements in the loop body requires the execution of instructions in all control paths to merge the multiple values of a variable. One general approach is to go through a sequence of code transformations: predication → vectorization(using one of the above methods) → remove vector predicates → remove scalar predicates^[2]. If the following code is used as an example to show these transformations;

```
for (i = 0; i < 1024; i++)
    if (A[i] > 0)
        C[i] = B[i];
    else
        D[i] = D[i-1];
```

- After predication

```
for (i = 0; i < 1024; i++)
{
    P = A[i] > 0;
    NP = !P;
    C[i] = B[i];      (P)
    D[i] = D[i-1];   (NP)
}
```

where (P) denotes a predicate guarding the statement.

- After vectorization

```
for (i = 0; i < 1024; i+=4)
{
    vP = A[i:i+3] > (0,0,0,0);
    vNP = vec_not(vP);
```

```

C[i:i+3] = B[i:i+3];      (vP)
(NP1,NP2,NP3,NP4) = vNP;
D[i+3] = D[i+2];          (NP4)
D[i+2] = D[i+1];          (NP3)
D[i+1] = D[i];            (NP2)
D[i]     = D[i-1];          (NP1)
}

```

- After removing vector predicates

```

for (i = 0; i < 1024; i+=4)
{
    vP = A[i:i+3] > (0,0,0,0);
    vNP = vec_not(vP);
    C[i:i+3] = vec_sel(C[i:i+3],B[i:i+3],vP);
    (NP1,NP2,NP3,NP4) = vNP;
    D[i+3] = D[i+2];          (NP4)
    D[i+2] = D[i+1];          (NP3)
    D[i+1] = D[i];            (NP2)
    D[i]     = D[i-1];          (NP1)
}

```

- After removing scalar predicates

```

for (i = 0; i < 1024; i+=4)
{
    vP = A[i:i+3] > (0,0,0,0);
    vNP = vec_not(vP);
    C[i:i+3] = vec_sel(C[i:i+3],B[i:i+3],vP);
    (NP1,NP2,NP3,NP4) = vNP;
    if (NP4) D[i+3] = D[i+2];
    if (NP3) D[i+2] = D[i+1];
    if (NP2) D[i+1] = D[i];
    if (NP1) D[i]     = D[i-1];
}

```

Reducing vectorization overhead in the presence of control flow

Having to execute the instructions in all control paths in vector code has been one of the major factors that slow down the vector code with respect to the scalar baseline. The more complex the control flow becomes and the more instructions are bypassed in the scalar code the larger the vectorization overhead grows. To reduce this vectorization overhead, vector branches can be inserted to bypass vector instructions similar to the way scalar branches bypass scalar instructions^[3]. Below, AltiVec predicates are used to show how this can be achieved.

- Scalar baseline (original code)

```

for (i = 0; i < 1024; i++)
{
    if (A[i] > 0)
    {
        C[i] = B[i];
    }
}

```

```

if (B[i] < 0)
    D[i] = E[i];
}
}

```

- After vectorization in the presence of control flow

```

for (i = 0; i < 1024; i+=4)
{
    vPA = A[i:i+3] > (0,0,0,0);
    C[i:i+3] = vec_sel(C[i:i+3],B[i:i+3],vPA);
    vT = B[i:i+3] < (0,0,0,0);
    vPB = vec_sel((0,0,0,0), vT, vPA);
    D[i:i+3] = vec_sel(D[i:i+3],E[i:i+3],vPB);
}

```

- After inserting vector branches

```

for (i = 0; i < 1024; i+=4)
    if (vec_any_gt(A[i:i+3], (0,0,0,0)))
    {
        vPA = A[i:i+3] > (0,0,0,0);
        C[i:i+3] = vec_sel(C[i:i+3],B[i:i+3],vPA);
        vT = B[i:i+3] < (0,0,0,0);
        vPB = vec_sel((0,0,0,0), vT, vPA);
        if (vec_any_ne(vPB, (0,0,0,0)))
            D[i:i+3] = vec_sel(D[i:i+3],E[i:i+3],vPB);
    }

```

There are two things to note in the final code with vector branches; First, the predicate defining instruction for vPA is also included within the body of the outer vector branch by using vec_any_gt. Second, the profitability of the inner vector branch for vPB depends on the conditional probability of vPB having false values in all fields given vPA has false values in all fields.

Consider an example where the outer branch in the scalar baseline is always taken, bypassing most instructions in the loop body. The intermediate case above, without vector branches, executes all vector instructions. The final code, with vector branches, executes both the comparison and the branch in vector mode, potentially gaining performance over the scalar baseline.

References

- [1] Larsen, S.; Amarasinghe, S. (2000). *Exploiting superword level parallelism with multimedia instruction sets*. "Proceedings of the ACM SIGPLAN conference on Programming language design and implementation". *ACM SIGPLAN Notices* **35** (5): 145–156. doi:10.1145/358438.349320.
- [2] Shin, J.; Hall, M. W.; Chame, J. (2005). "Superword-Level Parallelism in the Presence of Control Flow". *Proceedings of the international symposium on Code generation and optimization*. pp. 165–175. doi:10.1109/CGO.2005.33.
- [3] Shin, J. (2007). "Introducing Control Flow into Vectorized Code". *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. pp. 280–291. doi:10.1109/PACT.2007.41.

Superword Level Parallelism

Superword level parallelism is a vectorization technique based on loop unrolling and basic block vectorization. It is available in the gcc 4.3 compiler.

External links

- Links to Publication on Superword Level Parallelism [1]

References

[1] <http://www.cag.lcs.mit.edu/slp/>

Code generation

Code generation

In computer science, **code generation** is the process by which a compiler's **code generator** converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine (often a computer).

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the *backend*) needs to change from target to target. (For more information on compiler design, see Compiler.)

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three address code. Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program. (For example, a peephole optimization pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.)

Major tasks in code generation

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way. The generator may try to use faster instructions, use fewer instructions, exploit available registers, and avoid redundant computations.

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

- Instruction selection: which instructions to use.
- Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.
- Register allocation: the allocation of variables to processor registers.

Instruction selection is typically carried out by doing a recursive postorder traversal on the abstract syntax tree, matching particular tree configurations against templates; for example, the tree $W := \text{ADD}(X, \text{MUL}(Y, Z))$ might be transformed into a linear sequence of instructions by recursively generating the sequences for $t1 := X$ and $t2 := \text{MUL}(Y, Z)$, and then emitting the instruction $\text{ADD } W, t1, t2$.

In a compiler that uses an intermediate language, there may be two instruction selection stages — one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the instruction set of the target machine. This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding opcodes. However, if the compiler is actually a language translator (for example, one that converts Eiffel to C), then the second code-generation phase may involve *building* a tree from the linear intermediate code.

Runtime code generation

When code generation occurs at runtime, as in just-in-time compilation (JIT), it is important that the entire process be efficient with respect to space and time. For example, when regular expressions are interpreted and used to generate code at runtime, a non-deterministic finite state machine is often generated instead of a deterministic one, because usually the former can be created more quickly and occupies less memory space than the latter. Despite its generally generating less efficient code, JIT code generation can take advantage of profiling information that is available only at runtime.

Related concepts

The fundamental task of taking input in one language and producing output in a non-trivially different language can be understood in terms of the core transformational operations of formal language theory. Consequently, some techniques that were originally developed for use in compilers have come to be employed in other ways as well. For example, YACC (Yet Another Compiler Compiler) takes input in Backus-Naur form and converts it to a parser in C. Though it was originally created for automatic generation of a parser for a compiler, yacc is also often used to automate writing code that needs to be modified each time specifications are changed. (For example, see [1].)

Many integrated development environments (IDEs) support some form of automatic source code generation, often using algorithms in common with compiler code generators, although commonly less complicated. (See also: Program transformation, Data transformation.)

Reflection

In general, a syntax and semantic analyzer tries to retrieve the structure of the program from the source code, while a code generator uses this structural information (e.g., data types) to produce code. In other words, the former *adds* information while the latter *loses* some of the information. One consequence of this information loss is that reflection becomes difficult or even impossible. To counter this problem, code generators often embed syntactic and semantic information in addition to the code necessary for execution.

References

- [1] <http://www.artima.com/weblogs/viewpost.jsp?thread=152273>

Name mangling

In compiler construction, **name mangling** (also called **name decoration**) is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern programming languages.

It provides a way of encoding additional information in the name of a function, structure, class or another datatype in order to pass more semantic information from the compilers to linkers.

The need arises where the language allows different entities to be named with the same identifier as long as they occupy a different namespace (where a namespace is typically defined by a module, class, or explicit *namespace* directive).

Any object code produced by compilers is usually linked with other pieces of object code (produced by the same or another compiler) by a type of program called a linker. The linker needs a great deal of information on each program entity. For example, to correctly link a function it needs its name, the number of arguments and their types, and so on.

C name decoration in Microsoft Windows

Although name mangling is not generally required or used by languages that do not support function overloading (such as C and classic Pascal), they use it in some cases to provide additional information about a function. For example, compilers targeted at Microsoft Windows platforms support a variety of calling conventions, which determine the manner in which parameters are sent to subroutines and results returned. Because the different calling conventions are not compatible with one another, compilers mangle symbols with codes detailing which convention should be used.

The mangling scheme was established by Microsoft, and has been informally followed by other compilers including Digital Mars, Borland, and GNU gcc, when compiling code for the Windows platforms. The scheme even applies to other languages, such as Pascal, D, Delphi, Fortran, and C#. This allows subroutines written in those languages to call, or be called by, existing Windows libraries using a calling convention different from their default.

When compiling the following C examples:

```
int __cdecl f (int x) { return 0; }
int __stdcall g (int y) { return 0; }
int __fastcall h (int z) { return 0; }
```

32 bit compilers emit, respectively:

```
_f
_g@4
_h@4
```

In the `stdcall` and `fastcall` mangling schemes, the function is encoded as `_name@X` and `@name@X` respectively, where **X** is the number of bytes, in decimal, of the argument(s) in the parameter list (including those passed in registers, for `fastcall`). In the case of `cdecl`, the function name is merely prefixed by an underscore.

Note that the 64-bit convention on Windows (Microsoft C) is no leading underscore. This difference may in some rare cases lead to unresolved externals when porting such code to 64 bits. For example, Fortran code can use 'alias' to link against a C method by name as follows:

```
SUBROUTINE f()
!DEC$ ATTRIBUTES C, ALIAS:'_f' :: f
END SUBROUTINE
```

This will compile and link fine under 32 bits, but generate an unresolved external '_f' under 64 bits. One work around for this is to not use 'alias' at all (in which the method names typically need to be capitalized in C and Fortran), or to use the BIND option:

```
SUBROUTINE f() BIND(C,NAME="f")
END SUBROUTINE
```

Name mangling in C++

C++ compilers are the most widespread and yet least standard users of name mangling. The first C++ compilers were implemented as translators to C source code, which would then be compiled by a C compiler to object code; because of this, symbol names had to conform to C identifier rules. Even later, with the emergence of compilers which produced machine code or assembly directly, the system's linker generally did not support C++ symbols, and mangling was still required.

The C++ language does not define a standard decoration scheme, so each compiler uses its own. C++ also has complex language features, such as classes, templates, namespaces, and operator overloading, that alter the meaning of specific symbols based on context or usage. Meta-data about these features can be disambiguated by mangling (decorating) the name of a symbol. Because the name-mangling systems for such features are not standardized across compilers, few linkers can link object code that was produced by different compilers.

Simple example

Consider the following two definitions of `f()` in a C++ program:

```
int f (void) { return 1; }
int f (int) { return 0; }
void g (void) { int i = f(), j = f(0); }
```

These are distinct functions, with no relation to each other apart from the name. If they were natively translated into C with no changes, the result would be an error — C does not permit two functions with the same name. The C++ compiler therefore will encode the type information in the symbol name, the result being something resembling:

```
int __f_v (void) { return 1; }
int __f_i (int) { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }
```

Notice that `g()` is mangled even though there is no conflict; name mangling applies to **all** symbols.

Complex example

For a more complex example, we'll consider an example of a real-world name mangling implementation: that used by GNU GCC 3.x, and how it mangles the following example class. The mangled symbol is shown below the respective identifier name.

```
namespace wikipedia
{
    class article
    {
        public:
            std::string format (void);
            /* = _ZN9wikipedia7article6formatEv */
```

```

bool print_to (std::ostream&);

/* = _ZN9wikipedia7article8print_toERSo */

class wikilink
{
public:
    wikilink (std::string const& name);
    /* = _ZN9wikipedia7article8wikilinkC1ERKSS */
};

};

}

```

All mangled symbols begin with **_Z** (note that an underscore followed by a capital is a reserved identifier in C and C++, so conflict with user identifiers is avoided); for nested names (including both namespaces and classes), this is followed by **N**, then a series of <length, id> pairs (the length being the length of the next identifier), and finally **E**. For example, `wikipedia::article::format` becomes

```
_ZN·9wikipedia·7article·6format·E
```

For functions, this is then followed by the type information; as `format()` is a `void` function, this is simply **v**; hence:

```
_ZN·9wikipedia·7article·6format·E·v
```

For `print_to`, a standard type `std::ostream` (or more properly `std::basic_ostream<char, char_traits<char>>`) is used, which has the special alias **So**; a reference to this type is therefore **RSo**, with the complete name for the function being:

```
_ZN·9wikipedia·7article·8print_to·E·RSo
```

How different compilers mangle the same functions

There isn't a standard scheme by which even trivial C++ identifiers are mangled, and consequently different compiler vendors (or even different versions of the same compiler, or the same compiler on different platforms) mangle public symbols in radically different (and thus totally incompatible) ways. Consider how different C++ compilers mangle the same functions:

Compiler	<code>void h(int)</code>	<code>void h(int, char)</code>	<code>void h(void)</code>
Intel C++ 8.0 for Linux	<code>_Zlhi</code>	<code>_Zlhic</code>	<code>_Zlhv</code>
HP aC++ A.05.55 IA-64	<code>_Zlhi</code>	<code>_Zlhic</code>	<code>_Zlhv</code>
GCC 3.x and 4.x	<code>_Zlhi</code>	<code>_Zlhic</code>	<code>_Zlhv</code>
GCC 2.9x	<code>h__Fi</code>	<code>h__Fic</code>	<code>h__Fv</code>
HP aC++ A.03.45 PA-RISC	<code>h__Fi</code>	<code>h__Fic</code>	<code>h__Fv</code>
Microsoft VC++ v6/v7	<code>?h@@YAXH@Z</code>	<code>?h@@YAXHD@Z</code>	<code>?h@@YAXXZ</code>
Digital Mars C++	<code>?h@@YAXH@Z</code>	<code>?h@@YAXHD@Z</code>	<code>?h@@YAXXZ</code>
Borland C++ v3.1	<code>@h\$qi</code>	<code>@h\$qizc</code>	<code>@h\$qv</code>
OpenVMS C++ V6.5 (ARM mode)	<code>H__XI</code>	<code>H__XIC</code>	<code>H__XV</code>
OpenVMS C++ V6.5 (ANSI mode)	<code>CXX\$__7H__FI0ARG51T</code>	<code>CXX\$__7H__FIC26CDH77</code>	<code>CXX\$__7H__FV2CB06E8</code>
OpenVMS C++ X7.1 IA-64	<code>CXX\$__Z1HI2DSQ26A</code>	<code>CXX\$__Z1HIC2NP3LI4</code>	<code>CXX\$__Z1HV0BCA19V</code>

SunPro CC	<code>__1cBh6Fi_v_</code>	<code>__1cBh6Fic_v_</code>	<code>__1cBh6F_v_</code>
Tru64 C++ V6.5 (ARM mode)	<code>h__Xi</code>	<code>h__Xic</code>	<code>h__Xv</code>
Tru64 C++ V6.5 (ANSI mode)	<code>_7h__Fi</code>	<code>_7h__Fic</code>	<code>_7h__Fv</code>
Watcom C++ 10.6	<code>W?h\$n(i)v</code>	<code>W?h\$n(ia)v</code>	<code>W?h\$n()v</code>

Notes:

- The Compaq C++ compiler on OpenVMS VAX and Alpha (but not IA-64) and Tru64 has two name mangling schemes. The original, pre-standard scheme is known as ARM model, and is based on the name mangling described in the C++ Annotated Reference Manual (ARM). With the advent of new features in standard C++, particularly templates, the ARM scheme became more and more unsuitable — it could not encode certain function types, or produced identical mangled names for different functions. It was therefore replaced by the newer "ANSI" model, which supported all ANSI template features, but was not backwards compatible.
- On IA-64, a standard ABI exists (see external links), which defines (among other things) a standard name-mangling scheme, and which is used by all the IA-64 compilers. GNU GCC 3.x, in addition, has adopted the name mangling scheme defined in this standard for use on other, non-Intel platforms.

Handling of C symbols when linking from C++

The job of the common C++ idiom:

```
#ifdef __cplusplus
extern "C" {
#endif
/* ... */
#ifndef __cplusplus
}
#endif
```

is to ensure that the symbols following are "unmangled" — that the compiler emits a binary file with their names undecorated, as a C compiler would do. As C language definitions are unmangled, the C++ compiler needs to avoid mangling references to these identifiers.

For example, the standard strings library, `<string.h>` usually contains something resembling:

```
#ifdef __cplusplus
extern "C" {
#endif

void *memset (void *, int, size_t);
char *strcat (char *, const char *);
int strcmp (const char *, const char *);
char *strcpy (char *, const char *);

#ifndef __cplusplus
}
#endif
```

Thus, code such as:

```
if (strcmp(argv[1], "-x") == 0)
    strcpy(a, argv[2]);
```

```
else
    memset (a, 0, sizeof(a));
```

uses the correct, unmangled `strcmp` and `memset`. If the `extern` had not been used, the (SunPro) C++ compiler would produce code equivalent to:

```
if (__1cGstrcmp6Fpkcl_i_(argv[1], "-x") == 0)
    __1cGstrcpy6Fpcpkc_0_(a, argv[2]);
else
    __1cGmemset6FpviI_0_ (a, 0, sizeof(a));
```

Since those symbols do not exist in the C runtime library (*e.g.* `libc`), link errors would result.

Standardised name mangling in C++

While it is a relatively common belief that standardised name mangling in the C++ language would lead to greater interoperability between implementations, this is not really the case. Name mangling is only one of several application binary interface issues in a C++ implementation. Other ABI issues like exception handling, virtual table layout, structure padding, *etc.* cause differing C++ implementations to be incompatible. Further, requiring a particular form of mangling would cause issues for systems where implementation limits (*e.g.* length of symbols) dictate a particular mangling scheme. A standardised *requirement* for name mangling would also prevent an implementation where mangling was not required at all — for example, a linker which understood the C++ language.

The C++ standard therefore does not attempt to standardise name mangling. On the contrary, the *Annotated C++ Reference Manual* (also known as *ARM*, ISBN 0-201-51459-1, section 7.2.1c) actively encourages the use of different mangling schemes to prevent linking when other aspects of the ABI, such as exception handling and virtual table layout, are incompatible.

Real-world effects of C++ name mangling

Because C++ symbols are routinely exported from DLL and shared object files, the name mangling scheme is not merely a compiler-internal matter. Different compilers (or different versions of the same compiler, in many cases) produce such binaries under different name decoration schemes, meaning that symbols are frequently unresolved if the compilers used to create the library and the program using it employed different schemes. For example, if a system with multiple C++ compilers installed (*e.g.* GNU GCC and the OS vendor's compiler) wished to install the Boost C++ Libraries, it would have to be compiled twice — once for the vendor compiler and once for GCC.

It is good for safety purposes that compilers producing incompatible object codes (codes based on different ABIs, regarding *e.g.* classes and exceptions) use different name mangling schemes. This guarantees that these incompatibilities are detected at the linking phase, not when executing the software (which could lead to obscure bugs and serious stability issues).

For this reason name decoration is an important aspect of any C++-related ABI.

Name mangling in Java

The language, compiler, and .class file format were all designed together (and had object-orientation in mind from the start), so the primary problem solved by name mangling doesn't exist in implementations of the Java runtime. There are, however, cases where an analogous transformation and qualification of names is necessary.

Creating unique names for inner and anonymous classes

The scope of anonymous classes is confined to their parent class, so the compiler must produce a "qualified" public name for the inner class, to avoid conflict where other classes (inner or not) exist in the same namespace. Similarly, anonymous classes must have "fake" public names generated for them (as the concept of anonymous classes exists only in the compiler, not the runtime). So, compiling the following java program

```
public class foo {
    class bar {
        public int x;
    }

    public void zark () {
        Object f = new Object () {
            public String toString() {
                return "hello";
            }
        };
    }
}
```

will produce three .class files:

- **foo.class**, containing the main (outer) class *foo*
- **foo\$bar.class**, containing the named inner class *foo.bar*
- **foo\$1.class**, containing the anonymous inner class (local to method *foo.zark*)

All of these class names are valid (as \$ symbols are permitted in the JVM specification) and these names are "safe" for the compiler to generate, as the Java language definition prohibits \$ symbols in normal java class definitions.

Name resolution in Java is further complicated at runtime, as fully qualified class names are unique only inside a specific classloader instance. Classloaders are ordered hierarchically and each Thread in the JVM has a so called context class loader, so in cases where two different classloader instances contain classes with the same name, the system first tries to load the class using the root (or system) classloader and then goes down the hierarchy to the context class loader.

Java Native Interface

Java's native method support allows java language programs to call out to programs written in another language (generally either C or C++). There are two name-resolution concerns here, neither of which is implemented in a particularly standard manner:

- Java to native name translation
- normal C++ name mangling

Name mangling in Python

A Python programmer can explicitly designate that the name of an attribute within a class body should be mangled by using a name with two leading underscores and not more than one trailing underscore. For example, `__thing` will be mangled, as will `__thing` and `__thing_`, but `_thing__` and `_thing___` will not.

On encountering name mangled attributes, Python transforms these names by a single underscore and the name of the enclosing class, for example:

```
class Test(object):
    def __mangled_name(self):
        pass
    def normal_name(self):
        pass

print dir(Test)
```

will output:

```
['__Test__mangled_name',
'__doc__',
'__module__',
'normal_name']
```

Name mangling in Borland's Turbo Pascal / Delphi range

To avoid name mangling in Pascal, use:

```
exports
  myFunc name 'myFunc', myProc name 'myProc';
```

Name mangling in Free Pascal

Free Pascal supports function and operator overloading, thus it also uses name mangling to support these features. On the other hand, Free Pascal is capable of calling symbols defined in external modules created with another language and exporting its own symbols to be called by another language. For further information, consult Chapter 6.2^[1] and Chapter 7.1^[2] of Free Pascal Programmer's Guide^[3].

Name mangling in Objective-C

Essentially two forms of method exist in Objective-C, the class ("static") method, and the instance method. A method declaration in Objective-C is of the following form

```
+ method name: argument name1:parameter1 ...
- method name: argument name1:parameter1 ...
```

Class methods are signified by +, instance methods use -. A typical class method declaration may then look like:

```
+ (id) initWithFrame: (int) number andY: (int) number;
+ (id) new;
```

with instance methods looking like

```
- (id) value;
- (id) setValue: (id) newValue;
```

Each of these method declarations have a specific internal representation. When compiled, each method is named according to the following scheme for class methods:

```
_c_Class_methodname_name1-name2- ...
```

and this for instance methods:

```
_i_Class_methodname_name1-name2- ...
```

The colons in the Objective-C syntax are translated to underscores. So, the Objective-C class method `+ (id) initWithFrame: (int) number andY: (int) number;`, if belonging to the `Point` class would translate as `_c_Point_initWithX_andY_`, and the instance method (belonging to the same class) `- (id) value;` would translate to `_i_Point_value`.

Each of the methods of a class are labeled in this way. However, in order to look up a method that a class may respond to would be tedious if all methods are represented in this fashion. Each of the methods is assigned a unique symbol (such as an integer). Such a symbol is known as a *selector*. In Objective-C, one can manage selectors directly — they have a specific type in Objective-C — SEL.

During compilation, a table is built that maps the textual representation (such as `_i_Point_value`) to selectors (which are given a type SEL). Managing selectors is more efficient than manipulating the textual representation of a method. Note that a selector only matches a method's name, not the class it belongs to — different classes can have different implementations of a method with the same name. Because of this, implementations of a method are given a specific identifier too — these are known as implementation pointers, and are given a type also, IMP.

Message sends are encoded by the compiler as calls to the `objc_msgSend (id receiver, SEL selector, ...)` function, or one of its cousins, where `receiver` is the receiver of the message, and SEL determines the method to call. Each class has its own table that maps selectors to their implementations — the implementation pointer specifies where in memory the actual implementation of the method resides. There are separate tables for class and instance methods. Apart from being stored in the SEL to IMP lookup tables, the functions are essentially anonymous.

The SEL value for a selector does not vary between classes. This enables polymorphism.

The Objective-C runtime maintains information about the argument and return types of methods. However, this information is not part of the name of the method, and can vary from class to class.

Since Objective-C does not support namespaces, there is no need for mangling of class names (that do appear as symbols in generated binaries).

Name mangling in Fortran

Name mangling is also necessary in Fortran compilers, originally because the language is case insensitive. Further mangling requirements were imposed later in the evolution of the language because of the addition of modules and other features in the Fortran 90 standard. The case mangling, especially, is a common issue that must be dealt with in order to call Fortran libraries (such as LAPACK) from other languages (such as C).

Because of the case insensitivity, the name of a subroutine or function "FOO" must be converted to a canonical case and format by the Fortran compiler so that it will be linked in the same way regardless of case. Different compilers have implemented this in various ways, and no standardization has occurred. The AIX and HP-UX Fortran compilers convert all identifiers to lower case ("foo"), while the Cray Unicos Fortran compilers converted identifiers all upper case ("FOO"). The GNU g77 compiler converts identifiers to lower case plus an underscore ("foo_"), except that identifiers already containing an underscore ("FOO_BAR") have two underscores appended ("foo_bar__"), following a convention established by f2c. Many other compilers, including SGI's IRIX compilers, gfortran, and Intel's Fortran compiler, convert all identifiers to lower case plus an underscore ("foo_" and "foo_bar_").

Identifiers in Fortran 90 modules must be further mangled, because the same subroutine name may apply to different routines in different modules.

External links

- Linux Itanium ABI for C++^[4], including name mangling scheme.
- c++filt^[5] — filter to demangle encoded C++ symbols
- undname^[6] — msvc tool to demangle names.
- The Objective-C Runtime System^[7] — From Apple's *The Objective-C Programming Language 1.0*^[8]
- C++ Name Mangling/Demangling^[9] Quite detailed explanation of Visual C++ compiler name mangling scheme
- PHP UnDecorateSymbolName^[10] a php script that demangles Microsoft Visual C's function names.
- Calling conventions for different C++ compilers^[11] contains detailed description of name mangling schemes for various x86 C++ compilers
- Macintosh C/C++ ABI Standard Specification^[12]
- Mixing C and C++ Code^[13]
- Symbol management – 'Linkers and Loaders' by John R. Levine^[14]

References

- [1] <http://www.freepascal.org/docs-html/prog/progse21.html>
- [2] <http://www.freepascal.org/docs-html/prog/progse28.html>
- [3] <http://www.freepascal.org/docs-html/prog/prog.html>
- [4] <http://www.codesourcery.com/cxx-abi/abi.html#mangling>
- [5] <http://sources.redhat.com/binutils/docs-2.15/binutils/c-filt.html>
- [6] <http://msdn2.microsoft.com/en-us/library/5x49w699.aspx>
- [7] http://developer.apple.com/legacy/mac/library/documentation/Cocoa/Conceptual/OOPandObjC1/Articles/ocRuntimeSystem.html##apple_ref/doc/uid/TP40005191-CH9-CJBBBCHG
- [8] <http://developer.apple.com/legacy/mac/library/documentation/Cocoa/Conceptual/OOPandObjC1/Introduction/introObjectiveC.html>
- [9] <http://www.kegel.com/mangle.html#operators>
- [10] <http://sourceforge.net/projects/php-ms-demangle/>
- [11] http://www.agner.org/optimize/calling_conventions.pdf
- [12] http://developer.apple.com/tools/mpw-tools/compilers/docs/abi_spec.pdf
- [13] <http://www.parashift.com/c++-faq-lite/mixing-c-and-cpp.html>
- [14] <http://www.iecc.com/linker/linker05.html>

Register allocation

In compiler optimization, **register allocation** is the process of assigning a large number of target program variables onto a small number of CPU registers. Register allocation can happen over a basic block (*local register allocation*), over a whole function/procedure (*global register allocation*), or in-between functions as a calling convention (*interprocedural register allocation*).

Introduction

In many programming languages, the programmer has the illusion of allocating arbitrarily many variables. However, during compilation, the compiler must decide how to allocate these variables to a small, finite set of registers. Not all variables are in use (or "live") at the same time, so some registers may be assigned to more than one variable. However, two variables in use at the same time cannot be assigned to the same register without corrupting its value. Variables which cannot be assigned to some register must be kept in RAM and loaded in/out for every read/write, a process called *spilling*. Accessing RAM is significantly slower than accessing registers and slows down the execution speed of the compiled program, so an optimizing compiler aims to assign as many variables to registers as possible. *Register pressure* is the term used when there are fewer hardware registers available than would have been optimal; higher pressure usually means that more spills and reloads are needed.

In addition, programs can be further optimized by assigning the same register to a source and destination of a `move` instruction whenever possible. This is especially important if the compiler is using other optimizations such as SSA analysis, which artificially generates additional `move` instructions in the intermediate code.

Isomorphism to graph colorability

Through liveness analysis, compilers can determine which sets of variables are live at the same time, as well as variables which are involved in `move` instructions. Using this information, the compiler can construct a graph such that every vertex represents a unique variable in the program. *Interference edges* connect pairs of vertices which are live at the same time, and *preference edges* connect pairs of vertices which are involved in `move` instructions. Register allocation can then be reduced to the problem of K-coloring the resulting graph, where K is the number of registers available on the target architecture. No two vertices sharing an interference edge may be assigned the same color, and vertices sharing a preference edge should be assigned the same color if possible. Some of the vertices may be precolored to begin with, representing variables which must be kept in certain registers due to calling conventions or communication between modules. As graph coloring in general is NP-complete, so is register allocation. However, good algorithms exist which balance performance with quality of compiled code.

Iterated Register Coalescing

Register allocators have several types, with Iterated Register Coalescing (IRC) being a more common one. IRC was invented by Lal George and Andrew Appel in 1996, building off of earlier work by Gregory Chaitin. IRC works based on a few principles. First, if there are any non-move related vertices in the graph with degree less than K the graph can be simplified by removing those vertices, since once those vertices are added back in it is guaranteed that a color can be found for them (simplification). Second, two vertices sharing a preference edges whose adjacency sets combined have a degree less than K can be combined into a single vertex, by the same reasoning (coalescing). If neither of the two steps can simplify the graph, simplification can be run again on move-related vertices (freezing). Finally, if nothing else works, vertices can be marked for potential spilling and removed from the graph (spill). Since all of these steps reduce the degrees of vertices in the graph, vertices may transform from being high-degree (degree > K) to low-degree during the algorithm, enabling them to be simplified or coalesced. Thus, the stages of the algorithm are iterated to ensure aggressive simplification and coalescing. The pseudo-code is thus:

```

function IRC_color g K :
repeat
  if  $\exists v$  s.t. !moveRelated(v)  $\wedge$  degree(v) < K then simplify v
  else if  $\exists e$  s.t. cardinality(neighbors(first v)  $\cup$  neighbors(second v)) < K then coalesce e
  else if  $\exists v$  s.t. moveRelated(v) then deletePreferenceEdges v
  else if  $\exists v$  s.t. !precolored(v) then spill v
  else return
loop

```

The coalescing done in IRC is conservative, because aggressive coalescing may introduce spills into the graph. However, additional coalescing heuristics such as George coalescing may coalesce more vertices while still ensuring that no additional spills are added. Work-lists are used in the algorithm to ensure that each iteration of IRC requires sub-quadratic time.

Recent developments

Graph coloring allocators produce efficient code, but their allocation time is high. In cases of static compilation, allocation time is not a significant concern. In cases of dynamic compilation, such as just-in-time (JIT) compilers, fast register allocation is important. An efficient technique proposed by Poletto and Sarkar is linear scan allocation [1]. This technique requires only a single pass over the list of variable live ranges. Ranges with short lifetimes are assigned to registers, whereas those with long lifetimes tend to be spilled, or reside in memory. The results are on average only 12% less efficient than graph coloring allocators.

The linear scan algorithm follows:

1. Perform dataflow analysis to gather liveness information. Keep track of all variables' live intervals, the interval when a variable is live, in a list sorted in order of increasing start point (note that this ordering is free if the list is built when computing liveness.) We consider variables and their intervals to be interchangeable in this algorithm.
2. Iterate through liveness start points and allocate a register from the available register pool to each live variable.
3. At each step maintain a list of active intervals sorted by the end point of the live intervals. (Note that insertion sort into a balanced binary tree can be used to maintain this list at linear cost.) Remove any expired intervals from the active list and free the expired interval's register to the available register pool.
4. In the case where the active list is size R we cannot allocate a register. In this case add the current interval to the active pool without allocating a register. Spill the interval from the active list with the furthest end point. Assign the register from the spilled interval to the current interval or, if the current interval is the one spilled, do not change register assignments.

Cooper and Dasgupta recently developed a "lossy" Chaitin-Briggs graph coloring algorithm suitable for use in a JIT.^[2] The "lossy" moniker refers to the imprecision the algorithm introduces into the interference graph. This optimization reduces the costly graph building step of Chaitin-Briggs making it suitable for runtime compilation. Experiments indicate that this lossy register allocator outperforms linear scan on the majority of tests used.

"Optimal" register allocation algorithms based on Integer Programming have been developed by Goodwin and Wilken for regular architectures. These algorithms have been extended to irregular architectures by Kong and Wilken.

While the worst case execution time is exponential, the experimental results show that the actual time is typically of order $O(n^{2.5})$ of the number of constraints n .^[3]

The possibility of doing register allocation on SSA-form programs is a focus of recent research.^[4] The interference graphs of SSA-form programs are chordal, and as such, they can be colored in polynomial time.

References

- [1] <http://www.cs.ucla.edu/~palsberg/course/cs132/linearscan.pdf>
- [2] Cooper, Dasgupta, "Tailoring Graph-coloring Register Allocation For Runtime Compilation", <http://llvm.org/pubs/2006-04-04-CGO-GraphColoring.html>
- [3] Kong, Wilken, "Precise Register Allocation for Irregular Architectures", http://www.ece.ucdavis.edu/cerl/cerl_arch/irreg.pdf
- [4] Brisk, Hack, Palsberg, Pereira, Rastello, "SSA-Based Register Allocation", ESWEEK Tutorial <http://thedude.cc.gt.atl.ga.us/tutorials/1/>

Chaitin's algorithm

Chaitin's algorithm is a bottom-up, graph coloring register allocation algorithm that uses cost/degree as its spill metric. It is named after its designer, Gregory Chaitin. Chaitin's algorithm was the first register allocation algorithm that made use of coloring of the interference graph for both register allocations and spilling.

Chaitin's algorithm was presented on the 1982 SIGPLAN Symposium on Compiler Construction, and published in the symposium proceedings. It was extension of an earlier 1981 paper on the use of graph coloring for register allocation. Chaitin's algorithm formed the basis of a large section of research into register allocators.

References

- Gregory Chaitin *Register allocation and spilling via graph coloring* [1]

References

- [1] <http://portal.acm.org/citation.cfm?id=989403>

Rematerialization

Rematerialization or **remat** is a compiler optimization which saves time by recomputing a value instead of loading it from memory. It is typically tightly integrated with register allocation, where it is used as an alternative to spilling registers to memory. It was conceived by Preston Briggs, Keith D. Cooper, and Linda Torczon in 1992.

Traditional optimizations such as common subexpression elimination and loop invariant hoisting often focus on eliminating redundant computation. Since computation requires CPU cycles, this is usually a good thing, but it has the potentially devastating side effect that it can increase the live ranges of variables and create many new variables, resulting in spills during register allocation. Rematerialization is nearly the opposite: it decreases register pressure by increasing the amount of CPU computation. To avoid adding more computation time than necessary, rematerialization is done only when the compiler can be confident that it will be of benefit — that is, when a register spill to memory would otherwise occur.

Rematerialization works by keeping track of the expression used to compute each variable, using the concept of available expressions. Sometimes the variables used to compute a value are modified, and so can no longer be used to rematerialize that value. The expression is then said to no longer be available. Other criteria must also be fulfilled, for example a maximum complexity on the expression used to rematerialize the value; it would do no good to rematerialize a value using a complex computation that takes more time than a load. Usually the expression must also have no side effects.

External links

- P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization^[1]. *Proceedings of the SIGPLAN 92 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 27(7), p.311-321. July 1992. The CiteSeer page for the original paper.
- Mukta Punjani. Register Rematerialization in GCC^[2]. Discusses gcc's implementation of rematerialization.

References

- [1] <http://citeseer.ist.psu.edu/briggs92rematerialization.html>
[2] <http://gcc.fyrm.net/summit/2004/Register%20Rematerialization.pdf>

Sethi-Ullman algorithm

In computer science, the **Sethi–Ullman algorithm** is an algorithm named after Ravi Sethi and Jeffrey D. Ullman, its inventors, for translating abstract syntax trees into machine code that uses as few instructions as possible.

Overview

When generating code for arithmetic expressions, the compiler has to decide which is the best way to translate the expression in terms of number of instructions used as well as number of registers needed to evaluate a certain subtree. Especially in the case that free registers are scarce, the order of evaluation can be important to the length of the generated code, because different orderings may lead to larger or smaller numbers of intermediate values being spilled to memory and then restored. The Sethi–Ullman algorithm (also known as **Sethi–Ullman numbering**) fulfills the property of producing code which needs the least number of instructions possible as well as the least number of storage references (under the assumption that at the most commutativity and associativity apply to the operators used, but distributive laws i.e. $a * b + a * c = a * (b + c)$ do not hold). Please note that the algorithm succeeds as well if neither commutativity nor associativity hold for the expressions used, and therefore arithmetic transformations can not be applied.

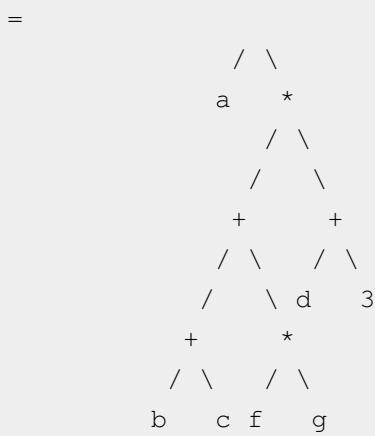
Simple Sethi–Ullman algorithm

The **simple Sethi–Ullman algorithm** works as follows (for a load-store architecture):

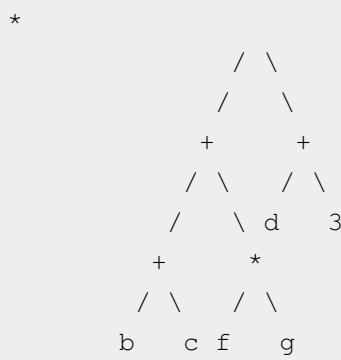
1. Traverse the abstract syntax tree in pre- or postorder
 1. For every non-constant leaf node, assign a 1 (i.e. 1 register is needed to hold the variable/field/etc.). For every constant leaf node (RHS of an operation – literals, values), assign a 0.
 2. For every non-leaf node n , assign the number of registers needed to evaluate the respective subtrees of n . If the number of registers needed in the left subtree (l) are not equal to the number of registers needed in the right subtree (r), the number of registers needed for the current node n is $\max(l, r)$. If $l == r$, then the number of registers needed for the current node is $l + 1$.
2. Code emission
 1. If the number of registers needed to compute the left subtree of node n is bigger than the number of registers for the right subtree, then the left subtree is evaluated first (since it may be possible that the one more register needed by the right subtree to save the result makes the left subtree spill). If the right subtree needs more registers than the left subtree, the right subtree is evaluated first accordingly. If both subtrees need equal as much registers, then the order of evaluation is irrelevant.

Example

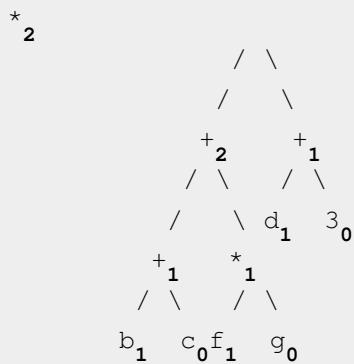
For an arithmetic expression $a = (b + c + f * g) * (d + 3)$, the abstract syntax tree looks like this:



To continue with the algorithm, we need only to examine the arithmetic expression $(b + c + f * g) * (d + 3)$, i.e. we only have to look at the right subtree of the assignment '=':



Now we start traversing the tree (in preorder for now), assigning the number of registers needed to evaluate each subtree (note that the last summand in the expression $(b + c + f * g) * (d + 3)$ is a constant):



From this tree it can be seen that we need 2 registers to compute the left subtree of the '*', but only 1 register to compute the right subtree. Nodes 'c' and 'g' do not need registers for the following reasons: If T is a tree leaf, then the number of registers to evaluate T is either 1 or 0 depending whether T is a left or a right subtree(since an operation such as add R1, A can handle the right component A directly without storing it into a register). Therefore we shall start to emit code for the left subtree first, because we might run into the situation that we only have 2 registers left to compute the whole expression. If we now computed the right subtree first (which needs only 1 register), we would then need a register to hold the result of the right subtree while computing the left subtree (which would still need 2 registers), therefore needing 3 registers concurrently. Computing the left subtree first needs 2 registers, but the result can be stored in 1, and since the right subtree needs only 1 register to compute, the evaluation of the expression can

do with only 2 registers left.

Advanced Sethi–Ullman algorithm

In an advanced version of the **Sethi–Ullman algorithm**, the arithmetic expressions are first transformed, exploiting the algebraic properties of the operators used.

References

- Sethi, Ravi; Ullman, Jeffrey D. (1970), "The Generation of Optimal Code for Arithmetic Expressions", *Journal of the Association for Computing Machinery* **17** (4): 715–728, doi:10.1145/321607.321620.

External links

- Code Generation for Trees ^[1]

References

- [1] <http://lambda.uta.edu/cse5317/fall02/notes/node40.html>

Data structure alignment

Data structure alignment is the way data is arranged and accessed in computer memory. It consists of two separate but related issues: *data alignment* and *data structure padding*. When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (e.g. 4 byte chunks on a 32-bit system). *Data alignment* means putting the data at a memory offset equal to some multiple of the word size, which increases the system's performance due to the way the CPU handles memory. To align the data, it may be necessary to insert some meaningless bytes between the end of the last data structure and the start of the next, which is *data structure padding*.

For example, when the computer's word size is 4 bytes (a byte meaning 8 bits), the data to be read should be at a memory offset which is some multiple of 4. When this is not the case, e.g. the data starts at the 14th byte instead of the 16th byte, then the computer has to read two 4-byte chunks and do some calculation before the requested data has been read, or it may generate an alignment fault. Even though the previous data structure ends at the 14th byte, the next data structure should start at the 16th byte. Two padding bytes are inserted between the two data structures to align the next data structure to the 16th byte.

Although data structure alignment is a fundamental issue for all modern computers, many computer languages and computer language implementations handle data alignment automatically. Ada,^[1] ^[2] certain C and C++ implementations, and assembly language allow at least partial control of data structure padding, which may be useful in certain special circumstances.

Definitions

A memory address a , is said to be *n-byte aligned* when n is a power of two and a is a multiple of n bytes. In this context a byte is the smallest unit of memory access, i.e. each memory address specifies a different byte. An n -byte aligned address would have $\log_2(n)$ least-significant zeros when expressed in binary.

The alternate wording *b-bit aligned* designates a $b/8$ byte aligned address (ex. 64-bit aligned is 8 bytes aligned).

A memory access is said to be *aligned* when the datum being accessed is n bytes long and the datum address is n -byte aligned. When a memory access is not aligned, it is said to be *misaligned*. Note that by definition byte memory accesses are always aligned.

A memory pointer that refers to primitive data that is n bytes long is said to be *aligned* if it is only allowed to contain addresses that are n -byte aligned, otherwise it is said to be *unaligned*. A memory pointer that refers to a data aggregate (a data structure or array) is *aligned* if (and only if) each primitive datum in the aggregate is aligned.

Note that the definitions above assume that each primitive datum is a power of two bytes long. When this is not the case (as with 80-bit floating-point on x86) the context influences the conditions where the datum is considered aligned or not.

Problems

A computer accesses memory by a single memory word at a time. As long as the memory word size is at least as large as the largest primitive data type supported by the computer, aligned accesses will always access a single memory word. This may not be true for misaligned data accesses.

If the highest and lowest bytes in a datum are not within the same memory word the computer must split the datum access into multiple memory accesses. This requires a lot of complex circuitry to generate the memory accesses and coordinate them. To handle the case where the memory words are in different memory pages the processor must either verify that both pages are present before executing the instruction or be able to handle a TLB miss or a page fault on any memory access during the instruction execution.

When a single memory word is accessed the operation is atomic, i.e. the whole memory word is read or written at once and other devices must wait until the read or write operation completes before they can access it. This may not be true for unaligned accesses to multiple memory words, e.g. the first word might be read by one device, both words written by another device and then the second word read by the first device so that the value read is neither the original value nor the updated value. Although such failures are rare, they can be very difficult to identify.

Architectures

RISC

Most RISC processors will generate an alignment fault when a load or store instruction accesses a misaligned address. This allows the operating system to emulate the misaligned access using other instructions. For example, the alignment fault handler might use byte loads or stores (which are always aligned) to emulate a larger load or store instruction.

Some architectures like MIPS have special unaligned load and store instructions. One unaligned load instruction gets the bytes from the memory word with the lowest byte address and another gets the bytes from the memory word with the highest byte address. Similarly, store-high and store-low instructions store the appropriate bytes in the higher and lower memory words respectively.

The Alpha architecture has a two-step approach to unaligned loads and stores. The first step is to load the upper and lower memory words into separate registers. The second step is to extract or modify the memory words using special low/high instructions similar to the MIPS instructions. An unaligned store is completed by storing the modified

memory words back to memory. The reason for this complexity is that the original Alpha architecture could only read or write 32-bit or 64-bit values. This proved to be a severe limitation that often led to code bloat and poor performance. To address this limitation, an extension called the Byte Word Extensions (BWX) was added to the original architecture. It consisted of instructions for byte and word loads and stores.

Because these instructions are larger and slower than the normal memory load and store instructions they should only be used when necessary. Most C and C++ compilers have an “unaligned” attribute that can be applied to pointers that need the unaligned instructions.

x86 and x86-64

While the x86 architecture originally did not require aligned memory access and still works without it, SSE2 instructions on x86 CPUs *do* require the data to be 128-bit (16-byte) aligned and there can be substantial performance advantages from using aligned data on these architectures. However, there are also instructions for unaligned access such as MOVDQU.

Compatibility

The advantage to supporting unaligned access is that it is easier to write compilers that do not need to align memory, at the expense of the cost of slower access. One way to increase performance in RISC processors which are designed to maximize raw performance is to require data to be loaded or stored on a word boundary. So though memory is commonly addressed by 8 bit bytes, loading a 32 bit integer or 64 bit floating point number would be required to start at every 64 bits on a 64 bit machine. The processor could flag a fault if it were asked to load a number which was not on such a boundary, but this would result in a slower call to a routine which would need to figure out which word or words contained the data and extract the equivalent value.

Data structure padding

Although the compiler (or interpreter) normally allocates individual data items on aligned boundaries, data structures often have members with different alignment requirements. To maintain proper alignment the translator normally inserts additional unnamed data members so that each member is properly aligned. In addition the data structure as a whole may be padded with a final unnamed member. This allows each member of an array of structures to be properly aligned.

Padding is only inserted when a structure member is followed by a member with a larger alignment requirement or at the end of the structure. By changing the ordering of members in a structure, it is possible to change the amount of padding required to maintain alignment. For example, if members are sorted by ascending or descending alignment requirements a minimal amount of padding is required. The minimal amount of padding required is always less than the largest alignment in the structure. Computing the maximum amount of padding required is more complicated, but is always less than the sum of the alignment requirements for all members minus twice the sum of the alignment requirements for the least aligned half of the structure members.

Although C and C++ do not allow the compiler to reorder structure members to save space, other languages might. It is also possible to tell most C and C++ compilers to "pack" the members of a structure to a certain level of alignment, e.g. "pack(2)" means align data members larger than a byte to a two-byte boundary so that any padding members are at most one byte long.

One use for such "packed" structures is to conserve memory. For example, a structure containing a single byte and a four-byte integer would require three additional bytes of padding. A large array of such structures would use 37.5% less memory if they are packed, although accessing each structure might take longer. This compromise may be considered a form of space-time tradeoff.

Although use of "packed" structures is most frequently used to conserve memory space, it may also be used to format a data structure for transmission using a standard protocol. However in this usage, care must also be taken to ensure that the values of the struct members are stored with the endianness required by the protocol (often network byte order), which may be different from the endianness used natively by the host machine.

Computing padding

The following formulas provide the number of padding bytes required to align the start of a data structure (where *mod* is the modulo operator):

```
# pseudo-code, see actual code below
padding = align - (offset mod align)
new offset = offset + padding = offset + align - (offset mod align)
```

For example, the padding to add to offset 0x59d for a structure aligned to every 4 bytes is 3. The structure will then start at 0x5a0, which is a multiple of 4. Note that when *offset* already is a multiple of *align*, taking the modulo of *align - (offset mod align)* is required to get a padding of 0.

If the alignment is a power of two, the modulo operation can be reduced to a bitwise boolean AND operation. The following formulas provide the new offset (where & is a bitwise AND and ~ a bitwise NOT):

```
padding = align - (offset & (align - 1)) = (~offset) & (align - 1)
new offset = (offset + align - 1) & ~ (align - 1)
```

Typical alignment of C structs on x86

Data structure members are stored sequentially in a memory so that in the structure below the member Data1 will always precede Data2 and Data2 will always precede Data3:

```
struct MyData
{
    short Data1;
    short Data2;
    short Data3;
};
```

If the type "short" is stored in two bytes of memory then each member of the data structure depicted above would be 2-byte aligned. Data1 would be at offset 0, Data2 at offset 2 and Data3 at offset 4. The size of this structure would be 6 bytes.

The type of each member of the structure usually has a default alignment, meaning that it will, unless otherwise requested by the programmer, be aligned on a pre-determined boundary. The following typical alignments are valid for compilers from Microsoft, Borland, and GNU when compiling for 32-bit x86:

- A **char** (one byte) will be 1-byte aligned.
- A **short** (two bytes) will be 2-byte aligned.
- An **int** (four bytes) will be 4-byte aligned.
- A **long** (four bytes) will be 4-byte aligned.
- A **float** (four bytes) will be 4-byte aligned.
- A **double** (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux (8-byte with *-malign-double* compile time option).
- A **long double** (twelve bytes) will be 4-byte aligned.
- Any **pointer** (four bytes) will be 4-byte aligned. (e.g.: **char***, **int***)

The only notable difference in alignment for a 64-bit system when compared to a 32-bit system is:

- A **long** (eight bytes) will be 8-byte aligned.
- A **double** (eight bytes) will be 8-byte aligned.
- A **long double** (Sixteen bytes) will be 16-byte aligned.
- Any **pointer** (eight bytes) will be 8-byte aligned.

Here is a structure with members of various types, totaling **8 bytes** before compilation:

```
struct MixedData
{
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

After compilation the data structure will be supplemented with padding bytes to ensure a proper alignment for each of its members:

```
struct MixedData /* After compilation in 32-bit x86 machine */
{
    char Data1; /* 1 byte */
    char Padding1[1]; /* 1 byte for the following 'short' to be aligned
on a 2 byte boundary
assuming that the address where structure begins is an even number */
    short Data2; /* 2 bytes */
    int Data3; /* 4 bytes - largest structure member */
    char Data4; /* 1 byte */
    char Padding2[3]; /* 3 bytes to make total size of the structure 12
bytes */
};
```

The compiled size of the structure is now 12 bytes. It is important to note that the last member is padded with the number of bytes required so that the total size of the structure should be a multiple of the largest alignment of any structure member (alignment(int) in this case, which = 4 on linux-32bit/gcc). In this case 3 bytes are added to the last member to pad the structure to the size of a 12 bytes (alignment(int) × 3).

```
struct FinalPad {
    double x;
    char n[1];
};
```

In this example the total size of the structure sizeof(FinalPad) = 12 not 9 (so that the size is a multiple of 4 (alignment(double) = 4 on linux-32bit/gcc)).

```
struct FinalPadShort {
    short s;
    char n[3];
};
```

In this example the total size of the structure `sizeof(FinalPadShort) = 6` not 5 (not 8 either) (so that the size is a multiple of 2 (alignment(`short`) = 2 on linux-32bit/gcc)).

It is possible to change the alignment of structures to reduce the memory they require (or to conform to an existing format) by reordering structure members or changing the compiler's alignment (or "packing") of structure members.

```
struct MixedData /* after reordering */
{
    char Data1;
    char Data4; /* reordered */
    short Data2;
    int Data3;
};
```

The compiled size of the structure now matches the pre-compiled size of **8 bytes**. Note that *Padding1[1]* has been replaced (and thus eliminated) by *Data4* and *Padding2[3]* is no longer necessary as the structure is already aligned to the size of a long word.

The alternative method of enforcing the *MixedData* structure to be aligned to a one byte boundary will cause the pre-processor to discard the pre-determined alignment of the structure members and thus no padding bytes would be inserted.

While there is no standard way of defining the alignment of structure members, some compilers use `#pragma` directives to specify packing inside source files. Here is an example:

```
#pragma pack(push) /* push current alignment to stack */
#pragma pack(1)      /* set alignment to 1 byte boundary */

struct MyPackedData
{
    char Data1;
    long Data2;
    char Data3;
};

#pragma pack(pop) /* restore original alignment from stack */
```

This structure would have a compiled size of **6 bytes** on a 32-bit system. The above directives are available in compilers from Microsoft[3], Borland, GNU[4] and many others.

Default packing and `#pragma pack`

On some Microsoft compilers, particularly for the RISC processor, there is a relationship between project default packing (the `/Zp` directive) and the `#pragma pack` directive which is unexpected for most people.

The `#pragma pack` directive can only be used to **reduce** the packing size of a structure from the project default packing. This leads to interoperability problems with library headers which use for example `#pragma pack(8)` if you set a project packing to smaller than this. The MSDN documentation^[5] states that if the `#pragma pack` packing is larger than or equal to the project packing, it will be ignored.

For this reason, one should never set a project packing to any value other than the default of 8 bytes, as it would break the `#pragma pack` directives used in library headers and result in binary incompatibilities between structures.

In particular, setting `/Zp1` breaks all `#pragma pack` directives other than `#pragma pack(1)`.

However, this limitation is not present when compiling for desktop processors, such as the x86 architecture .

Allocating memory aligned to cache lines

It would be beneficial to allocate memory aligned to cache lines. If an array is partitioned for more than one thread to operate on, having the sub-array boundaries unaligned to cache lines could lead to performance degradation. Here is an example to allocate memory (double array of size 10) aligned to cache of 64 bytes.

```
#include <stdlib.h>
double *foo(void) {
    double *var; //create array of size 10
    int     ok;

    ok = posix_memalign((void**)&var, 64, 10*sizeof(double));
    if(ok != 0)
        return NULL;

    return var;
}
```

Hardware significance of alignment requirements

Alignment concerns can affect areas much larger than a C structure when the purpose is the efficient mapping of that area through a hardware address translation mechanism (PCI remapping, operation of a MMU).

For instance, on a 32bit operating system a 4KB page is not just an arbitrary 4KB chunk of data. Instead, it is usually a region of memory that's aligned on a 4KB boundary. This is because aligning a page on a page-sized boundary lets the hardware map a virtual address to a physical address by substituting the higher bits in the address, rather than doing complex arithmetic.

Example: Assume that we have a TLB mapping of virtual address 0x2fcf7000 to physical address 0x12345000. (Note that both these addresses are aligned at 4KB boundaries.) Accessing data located at virtual address va=0x2fcf7abc causes a TLB resolution of 0x2fcf7 to 0x12345 to issue a physical access to pa=0x12345abc. Here, the 20/12bit split luckily match the hexadecimal representation split at 5/3 digits. The hardware can implement this translation by simply combining the first 20 bits of the physical address (0x12345) and the last 12 bits of the virtual address (0xabc). This is also referred to as virtually indexed(abc) physically tagged(12345).

A block of data of size $2^{n+1}-1$ always has one sub-block of size 2^n aligned on 2^n bytes.

This is how a dynamic allocator that has no knowledge of alignment, can be used to provide aligned buffers, at the price of a factor two in data loss.

Example: get a 12bit aligned 4KBytes buffer with malloc()

```
// unaligned pointer to large area
void *up=malloc((1<<13)-1);
// well aligned pointer to 4KBytes
void *ap=aligntonext(up,12);
```

where aligntonext() is meant as:

move p to the right until next well aligned address **if**
not correct already. A possible implementation is

```
// PSEUDOCODE assumes uint32_t p,bits; for readability
```

```
// --- not typesafe, not side-effect safe
#define alignto(p,bits) (p>>bits<<bits)
#define aligntonext(p,bits) alignto((p+(1<<bits)-1),bits)
```

References

- [1] "Ada Representation Clauses and Pragmas" (http://www.adacore.com/wp-content/files/auto_update/gnat-unw-docs/html/gnat_rm_7.html). . Retrieved 2011-01-11.
- [2] "F.8 Representation Clauses" (<http://docs.sun.com/app/docs/doc/801-4862/6hvbkina?l=en&a=view>). *SPARCompiler Ada Programmer's Guide*. . Retrieved 2011-01-11.
- [3] [http://msdn.microsoft.com/en-us/library/2e70t5y1\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/2e70t5y1(VS.80).aspx)
- [4] http://gcc.gnu.org/onlinedocs/gcc/Structure_002dPacking-Pragmas.html
- [5] "Working with Packing Structures" (<http://msdn.microsoft.com/en-us/library/ms253935.aspx>). *MSDN Library*. Microsoft. 2007-07-09. . Retrieved 2011-01-11.

Further reading

- Bryant, Randal E.; David, O'Hallaron (2003), *Computer Systems: A Programmer's Perspective* (<http://csapp.cs.cmu.edu/>) (2003 ed.), Upper Saddle River, NJ: Pearson Education, ISBN 0-13-034074-X

External links

- IBM developerWorks article on data alignment (<http://www.ibm.com/developerworks/library/pa-dalign/>)
- MSDN article on data alignment (<http://msdn2.microsoft.com/en-us/library/ms253949.aspx>)
- Article on data alignment and data portability (<http://www.codesynthesis.com/~boris/blog/2009/04/06/cxx-data-alignment-portability/>)
- Byte Alignment and Ordering (<http://www.eventhelix.com/RealtimeMantra/ByteAlignmentAndOrdering.htm>)
- Intel Itanium Architecture Software Developer's Manual (<http://developer.intel.com/design/itanium/manuals/245317.htm>)
- PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors (<http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2>)

Instruction selection

In computer science, **instruction selection** is the stage of a compiler backend that transforms its tree-based middle-level intermediate representation (IR) into a low-level IR very close to its final target language. In a typical compiler, it precedes both instruction scheduling and register allocation, so its output IR has an infinite set of pseudoregisters and may still be subject to peephole optimization; otherwise, it closely resembles the target machine code, bytecode, or assembly language. It works by "covering" the intermediate representation with as few *tiles* as possible. A tile is a template that matches a portion of the IR tree and can be implemented with a single target instruction.

Approach

A basic approach in instruction selection is to use some templates for translation of each instruction in an intermediate representation. But naïve use of templates leads to inefficient code in general. Additional attention needs to be paid to avoid duplicated memory access by reordering and merging instructions and promote the usage of registers.

For example, see the following sequence of intermediate instructions:

```
t1 = a  
t2 = b  
t3 = t1 + t2  
a = t3  
b = t1
```

A good tiling for the x86 architecture is a succinct set of instructions:

```
XCHG EAX, b  
ADD a, EAX
```

Typically, instruction selection is implemented with a backwards dynamic programming algorithm which computes the "optimal" tiling for each point starting from the end of the program and based from there. Instruction selection can also be implemented with a greedy algorithm that chooses a local optimum at each step.

The code that performs instruction selection is usually automatically generated from a list of valid patterns. Various generator programs differ in the amount of analysis that they perform while they run, rather during the compiler's instruction selection phase. An example of generator program for instruction selection is BURG.

Lowest common denominator strategy

The *lowest common denominator strategy* is an instruction selection technique used on platforms where Processor Supplementary Instructions exist to make executable programs portable across a wide range of computers. Under a lowest common denominator strategy, the default behaviour of the compiler is to build for the lowest common architecture. Use of any available processor extension is switched off by default, unless explicitly switched on by command line switches.

The use of a lowest common denominator strategy means that Processor Supplementary Instructions and Processor Supplementary Capabilities features are not used by default.

External links

- Alternative ways of supporting different generations of computer [1]

References

[1] <http://markhobley.yi.org/programming/generations.html>

Instruction scheduling

In computer science, **instruction scheduling** is a compiler optimization used to improve instruction-level parallelism, which improves performance on machines with instruction pipelines. Put more simply, without changing the meaning of the code, it tries to

- Avoid pipeline stalls by rearranging the order of instructions.
- Avoid illegal or semantically ambiguous operations (typically involving subtle instruction pipeline timing issues or non-interlocked resources.)

The pipeline stalls can be caused by structural hazards (processor resource limit), data hazards (output of one instruction needed by another instruction) and control hazards (branching).

Data hazards

Instruction scheduling is typically done on a single basic block. In order to determine whether rearranging the block's instructions in a certain way preserves the behavior of that block, we need the concept of a *data dependency*. There are three types of dependencies, which also happen to be the three data hazards:

1. Read after Write (RAW or "True"): Instruction 1 writes a value used later by Instruction 2. Instruction 1 must come first, or Instruction 2 will read the old value instead of the new.
2. Write after Read (WAR or "Anti"): Instruction 1 reads a location that is later overwritten by Instruction 2. Instruction 1 must come first, or it will read the new value instead of the old.
3. Write after Write (WAW or "Output"): Two instructions both write the same location. They must occur in their original order.

Technically, there is a fourth type, Read after Read (RAR or "Input"): Both instructions read the same location. Input dependence does not constrain the execution order of two statements, but it is useful in scalar replacement of array elements.

To make sure we respect the three types of dependencies, we construct a dependency graph, which is a directed graph where each vertex is an instruction and there is an edge from I_1 to I_2 if I_1 must come before I_2 due to a dependency. If loop-carried dependencies are left out, the dependency graph is a directed acyclic graph. Then, any topological sort of this graph is a valid instruction schedule. The edges of the graph are usually labelled with the **latency** of the dependence. This is the number of clock cycles that needs to elapse before the pipeline can proceed with the target instruction without stalling.

Algorithms

The simplest algorithm to find a topological sort is frequently used and is known as list scheduling. Conceptually, it repeatedly selects a source of the dependency graph, appends it to the current instruction schedule and removes it from the graph. This may cause other vertices to be sources, which will then also be considered for scheduling. The algorithm terminates if the graph is empty.

To arrive at a good schedule, stalls should be prevented. This is determined by the choice of the next instruction to be scheduled. A number of heuristics are in common use:

- The processor resources used by the already scheduled instructions are recorded. If a candidate uses a resource that is occupied, its priority will drop.
- If a candidate is scheduled closer to its predecessors than the associated latency its priority will drop.
- If a candidate lies on the critical path of the graph, its priority will rise. This heuristic provides some form of look-ahead in an otherwise local decision process.
- If choosing a candidate will create many new sources, its priority will rise. This heuristic tends to generate more freedom for the scheduler.

The phase order of Instruction Scheduling

Instruction scheduling may be done either before or after register allocation or both before and after it. The advantage of doing it before register allocation is that this results in maximum parallelism. The disadvantage of doing it before register allocation is that this can result in the register allocator needing to use a number of registers exceeding those available. This will cause spill/fill code to be introduced which will reduce the performance of the section of code in question.

If the architecture being scheduled has instruction sequences that have potentially illegal combinations (due to a lack of instruction interlocks) the instructions must be scheduled after register allocation. This second scheduling pass will also improve the placement of the spill/fill code.

If scheduling is only done after register allocation then there will be false dependencies introduced by the register allocation that will limit the amount of instruction motion possible by the scheduler.

Types of Instruction Scheduling

There are several types of instruction scheduling:

1. Local (Basic Block) Scheduling: instructions can't move across basic block boundaries.
2. Global scheduling: instructions can move across basic block boundaries.
3. Modulo Scheduling: another name for software pipelining, which is a form of instruction scheduling that interleaves different iterations of a loop.
4. Trace scheduling: the first practical approach for global scheduling, trace scheduling tries to optimize the control flow path that is executed most often.
5. Superblock scheduling: a simplified form of trace scheduling which does not attempt to merge control flow paths at trace "side entrances". Instead, code can be implemented by more than one schedule, vastly simplifying the code generator.

Software pipelining

In computer science, **software pipelining** is a technique used to optimize loops, in a manner that parallels hardware pipelining. Software pipelining is a type of out-of-order execution, except that the reordering is done by a compiler (or in the case of hand written assembly code, by the programmer) instead of the processor. Some computer architectures have explicit support for software pipelining, notably Intel's IA-64 architecture.

It is important to distinguish *software pipelining* which is a target code technique for overlapping loop iterations, from *modulo scheduling*, the currently most effective known compiler technique for generating software pipelined loops. Software pipelining has been known to assembly language programmers of machines with instruction level parallelism since such architectures existed. Effective compiler generation of such code dates to the invention of modulo scheduling by Rau and Glaeser^[1]. Lam showed that special hardware is unnecessary for effective modulo scheduling. Her technique, *modulo renaming* is widely used in practice^[2]. Gao et al. formulated optimal software pipelining in integer linear programming, culminating in validation of advanced heuristics in an evaluation paper^[3]. This paper has a good set of references on the topic.

Example

Consider the following loop:

```
for (i = 1) to bignumbr
    A(i)
    B(i)
    C(i)
end
```

In this example, let $A(i)$, $B(i)$, $C(i)$, be instructions, each operating on data i , that are dependent on each other. In other words, $A(i)$ must complete before $B(i)$ can start. For example, A could load data from memory into a register, B could perform some arithmetic operation on the data, and C could store the data back into memory. However, let there be no dependence between operations for different values of i . In other words, $A(2)$ can begin before $A(1)$ finishes.

Without software pipelining, the operations will execute in the following sequence:

```
A(1) B(1) C(1) A(2) B(2) C(2) A(3) B(3) C(3) ...
```

Assume that each instruction takes 3 clock cycles to complete (ignore for the moment the cost of the looping control flow). Also assume (as is the case on most modern systems) that an instruction can be dispatched every cycle, as long as it has no dependencies on an instruction that is already executing. In the unpipelined case, each iteration thus takes 7 cycles to complete ($3 + 3 + 1$, because $A(i+1)$ does not have to wait for $C(i)$).

Now consider the following sequence of instructions (with software pipelining):

```
A(1) A(2) A(3) B(1) B(2) B(3) C(1) C(2) C(3) ...
```

It can be easily verified that an instruction can be dispatched each cycle, which means that the same 3 iterations can be executed in a total of 9 cycles, giving an average of 3 cycles per iteration.

Implementation

Software pipelining is often used in combination with loop unrolling, and this combination of techniques is often a far better optimization than loop unrolling alone. In the example above, we could write the code as follows (assume for the moment that `bignumber` is divisible by 3):

```
for i = 1 to (bignumber - 2) step 3
    A(i)
    A(i+1)
    A(i+2)
    B(i)
    B(i+1)
    B(i+2)
    C(i)
    C(i+1)
    C(i+2)
end
```

Of course, matters are complicated if (as is usually the case) we can't guarantee that the number of iterations will be divisible by the number of iterations we unroll. See the article on loop unrolling for more on solutions to this problem. It should be noted, however, that software pipelining prevents the use of Duff's device, a widely known and efficient solution to this problem.

In the general case, loop unrolling may not be the best way to implement software pipelining. Consider a loop containing instructions with a high latency. For example, the following code:

```
for i = 1 to bignumber
    A(i) ; 3 cycle latency
    B(i) ; 3
    C(i) ; 12 (perhaps a floating point operation)
    D(i) ; 3
    E(i) ; 3
    F(i) ; 3
end
```

would require 12 iterations of the loop to be unrolled to avoid the bottleneck of instruction `C`. This means that the code of the loop would increase by a factor of 12 (which not only affects memory usage, but can also affect cache performance, *see code bloat*). Even worse, the prolog (code before the loop for handling the case of `bignumber` not divisible by 12) will likely be even larger than the code for the loop, and very probably inefficient because software pipelining cannot be used in this code (at least not without a significant amount of further code bloat). Furthermore, if `bignumber` is expected to be moderate in size compared to the number of iterations unrolled (say 10-20), then the execution will spend most of its time in this inefficient prolog code, rendering the software pipelining optimization ineffectual.

Here is an alternative implementation of software pipelining for our example (the *prolog* and *postlog* will be explained later):

```
prolog
for i = 1 to (bignumber - 6)
    A(i+6)
    B(i+5)
    C(i+4)
```

```

D(i+2) ; note that we skip i+3
E(i+1)
F(i)
end
postlog

```

Before getting to the prolog and postlog, which handle iterations at the beginning and end of the loop, let's verify that this code does the same thing as the original for iterations in the middle of the loop. Specifically, consider iteration 7 in the original loop. The first iteration of the pipelined loop will be the first iteration that includes an instruction from iteration 7 of the original loop. The sequence of instructions is:

```

Iteration 1: A(7) B(6) C(5) D(3) E(2) F(1)
Iteration 2: A(8) B(7) C(6) D(4) E(3) F(2)
Iteration 3: A(9) B(8) C(7) D(5) E(4) F(3)
Iteration 4: A(10) B(9) C(8) D(6) E(5) F(4)
Iteration 5: A(11) B(10) C(9) D(7) E(6) F(5)
Iteration 6: A(12) B(11) C(10) D(8) E(7) F(6)
Iteration 7: A(13) B(12) C(11) D(9) E(8) F(7)

```

However, unlike the original loop, the pipelined version avoid the bottleneck at instruction C. Note that there are 12 instructions between C(7) and the dependent instruction D(7), which means that the latency cycles of instruction C(7) are used for other instructions instead of being wasted.

The prolog and postlog handle iterations at the beginning and end of the loop. Here is a possible prolog for our example above:

```

; loop prolog (arranged on lines for clarity)
A(1)
A(2), B(1)
A(3), B(2), C(1)
A(4), B(3), C(2)
A(5), B(4), C(3), D(1)
A(6), B(5), C(4), D(2), E(1)

```

Each line above corresponds to an iteration of the pipelined loop, with instructions for iterations that have not yet begun removed. The postlog would look similar.

Difficulties of implementation

The requirement of a prologue and epilogue is one of the major difficulties of implementing software pipelining. Note that the prologue in this example is 18 instructions, 3 times as large as the loop itself. The postlog would also be 18 instructions. In other words, the prologue and epilogue together are *6 times as large as the loop itself*. While still better than attempting loop unrolling for this example, software pipelining requires a trade-off between speed and memory usage. Keep in mind, also, that if the code bloat is too large, it will affect speed anyway via a decrease in cache performance.

A further difficulty is that on many architectures, most instructions use a register as an argument, and that the specific register to use must be hard-coded into the instruction. In other words, on many architectures, it is impossible to code such an instruction as "multiply the contents of register X and register Y and put the result in register Z", where X, Y, and Z are numbers taken from other registers or memory. This has often been cited as a reason that software pipelining cannot be effectively implemented on conventional architectures.

In fact, Monica Lam presents an elegant solution to this problem in her thesis, *A Systolic Array Optimizing Compiler* (1989) (ISBN 0-89838-300-5). She calls it *Modulo Renaming*. The trick is to replicate the body of the loop after it has been scheduled, allowing different registers to be used for different values of the same variable when they have to be live at the same time. For the simplest possible example, let's suppose that $A(i)$ and $B(i)$ can be issued in parallel and that the latency of the latter is 2 cycles. The pipelined body could then be:

```
A(i+2); B(i)
```

Register allocation of this loop body runs into the problem that the result of $A(i+2)$ must stay live for two iterations. Using the same register for the result of $A(i+2)$ and the input of $B(i)$ will result in incorrect results.

However, if we replicate the scheduled loop body, the problem is solved:

```
A(i+2); B(i)
A(i+3); B(i+1)
```

Now a separate register can be allocated to the results of $A(i+2)$ and $A(i+3)$. To be more concrete:

```
r1 = A(i+2); B(i) = r1
r2 = A(i+3); B(i+1) = r2
i = i + 2 // Just to be clear
```

On the assumption that each instruction bundle reads its input registers before writing its output registers, this code is correct. At the start of the replicated loop body, $r1$ holds the value of $A(i+2)$ from the previous replicated loop iteration. Since i has been incremented by 2 in the meantime, this is actually the value of $A(i)$ in this replicated loop iteration.

Of course, code replication increases code size and cache pressure just as the prologue and epilogue do. Nevertheless, for loops with large trip counts on architectures with enough instruction level parallelism, the technique easily performs well enough to be worth any increase in code size.

IA-64 implementation

Intel's IA-64 architecture provides an example of an architecture designed with the difficulties of software pipelining in mind. Some of the architectural support for software pipelining includes:

- A "rotating" register bank; instructions can refer to a register number that is redirected to a different register each iteration of the loop (eventually looping back around to the beginning). This makes the extra instructions inserted in the previous example unnecessary.
- Predicates (used to "predicate" instructions; see *Branch predication*) that take their value from special looping instructions. These predicates turn on or off certain instructions in the loop, making a separate prolog and postlog unnecessary.

References

- [1] B.R. Rau and C.D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", In *Proceedings of the Fourteenth Annual Workshop on Microprogramming (MICRO-14)*, December 1981, pages 183-198
- [2] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI 88)*, July 1988 pages 318-328. Also published as ACM SIGPLAN Notices 23(7).
- [3] J. Ruttenberg, G.R. Gao, A. Stouchin, and W. Lichtenstein, "Software pipelining showdown: optimal vs. heuristic methods in a production compiler", In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996, pages 1-11. Also published as ACM SIGPLAN Notices 31(5).

Trace scheduling

Trace scheduling is an optimization technique used in compilers for computer programs.

A compiler often can, by rearranging its generated machine instructions for faster execution, improve program performance. Trace scheduling is one of many known techniques for doing so.

Trace scheduling was originally developed for Very Long Instruction Word, or VLIW machines, and is a form of global code motion. It works by converting a loop to long straight-line code sequence using loop unrolling and static branch prediction. This process separates out "unlikely" code and adds handlers for exits from trace. The goal is to have the most common case executed as a sequential set of instructions without branches.

Just-in-time compilation

In computing, **just-in-time compilation (JIT)**, also known as **dynamic translation**, is a method to improve the runtime performance of computer programs. Traditionally, computer programs had two modes of runtime operation, either interpreted or static (ahead-of-time) compilation. Interpreted code is translated from a high-level language to a machine code continuously during every execution, whereas statically compiled code is translated into machine code before execution, and only requires this translation once.

JIT compilers represent a hybrid approach, with translation occurring continuously, as with interpreters, but with caching of translated code to minimize performance degradation. It also offers other advantages over statically compiled code at development time, such as handling of late-bound data types and the ability to enforce security guarantees.

JIT builds upon two earlier ideas in run-time environments: *bytecode compilation* and *dynamic compilation*. It converts code at *runtime* prior to executing it natively, for example bytecode into native machine code.

Several modern runtime environments, such as Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution.

Overview

In a bytecode-compiled system, source code is translated to an intermediate representation known as bytecode. Bytecode is not the machine code for any particular computer, and may be portable among computer architectures. The bytecode may then be interpreted by, or run on, a virtual machine. The JIT compiler reads the bytecodes in many sections (or in full rarely) and compiles them interactively into machine language so the program can run faster. Java performs runtime checks on various sections of the code and this is the reason the entire code is not compiled at once.^[1] This can be done per-file, per-function or even on any arbitrary code fragment; the code can be compiled when it is about to be executed (hence the name "just-in-time"), and then cached and reused later without needing to be recompiled.

In contrast, a traditional *interpreted virtual machine* will simply interpret the bytecode, generally with much lower performance. Some *interpreters* even interpret source code, without the step of first compiling to bytecode, with even worse performance. *Statically compiled code* or *native code* is compiled prior to deployment. A *dynamic compilation environment* is one in which the compiler can be used during execution. For instance, most Common Lisp systems have a `compile` function which can compile new functions created during the run. This provides many of the advantages of JIT, but the programmer, rather than the runtime, is in control of what parts of the code are compiled. This can also compile dynamically generated code, which can, in many scenarios, provide substantial performance advantages over statically compiled code, as well as over most JIT systems.

A common goal of using JIT techniques is to reach or surpass the performance of static compilation, while maintaining the advantages of bytecode interpretation: Much of the "heavy lifting" of parsing the original source code and performing basic optimization is often handled at compile time, prior to deployment: compilation from bytecode to machine code is much faster than compiling from source. The deployed bytecode is portable, unlike native code. Since the runtime has control over the compilation, like interpreted bytecode, it can run in a secure sandbox. Compilers from bytecode to machine code are easier to write, because the portable bytecode compiler has already done much of the work.

JIT code generally offers far better performance than interpreters. In addition, it can in some cases offer better performance than static compilation, as many optimizations are only feasible at run-time:

1. The compilation can be optimized to the targeted CPU and the operating system model where the application runs. For example JIT can choose SSE2 CPU instructions when it detects that the CPU supports them. To obtain this level of optimization specificity with a static compiler, one must either compile a binary for each intended platform/architecture, or else include multiple versions of portions of the code within a single binary.
2. The system is able to collect statistics about how the program is actually running in the environment it is in, and it can rearrange and recompile for optimum performance. However, some static compilers can also take profile information as input.
3. The system can do global code optimizations (e.g. inlining of library functions) without losing the advantages of dynamic linking and without the overheads inherent to static compilers and linkers. Specifically, when doing global inline substitutions, a static compilation process may need run-time checks and ensure that a virtual call would occur if the actual class of the object overrides the inlined method, and boundary condition checks on array accesses may need to be processed within loops. With just-in-time compilation in many cases this processing can be moved out of loops, often giving large increases of speed.
4. Although this is possible with statically compiled garbage collected languages, a bytecode system can more easily rearrange executed code for better cache utilization.

Startup delay and optimizations

JIT typically causes a slight delay in initial execution of an application, due to the time taken to load and compile the bytecode. Sometimes this delay is called "startup time delay". In general, the more optimization JIT performs, the better the code it will generate, but the initial delay will also increase. A JIT compiler therefore has to make a trade-off between the compilation time and the quality of the code it hopes to generate. However, it seems that much of the startup time is sometimes due to IO-bound operations rather than JIT compilation (for example, the `rt.jar` class data file for the Java Virtual Machine is 40 MB and the JVM must seek a lot of data in this contextually huge file).^[2]

One possible optimization, used by Sun's HotSpot Java Virtual Machine, is to combine interpretation and JIT compilation. The application code is initially interpreted, but the JVM monitors which sequences of bytecode are frequently executed and translates them to machine code for direct execution on the hardware. For bytecode which is executed only a few times, this saves the compilation time and reduces the initial latency; for frequently executed bytecode, JIT compilation is used to run at high speed, after an initial phase of slow interpretation. Additionally, since a program spends most time executing a minority of its code, the reduced compilation time is significant.

Finally, during the initial code interpretation, execution statistics can be collected before compilation, which helps to perform better optimization.^[3]

The correct tradeoff can vary due to circumstances. For example, Sun's Java Virtual Machine has two major modes—client and server. In client mode, minimal compilation and optimization is performed, to reduce startup time. In server mode, extensive compilation and optimization is performed, to maximize performance once the application is running by sacrificing startup time. Other Java just-in-time compilers have used a runtime measurement of the number of times a method has executed combined with the bytecode size of a method as a heuristic to decide when to compile.^[4] Still another uses the number of times executed combined with the detection of loops.^[5] In general, it is much harder to accurately predict which methods to optimize in short-running applications than in long-running ones.^[6]

Native Image Generator (Ngen) by Microsoft is another approach at reducing the initial delay.^[7] Ngen pre-compiles (or "pre-jits") bytecode in a Common Intermediate Language image into machine native code. As a result, no runtime compilation is needed. .NET framework 2.0 shipped with Visual Studio 2005 runs Ngen on all of the Microsoft library DLLs right after the installation. Pre-jitting provides a way to improve the startup time. However, the quality of code it generates might not be as good as the one that is jitted, for the same reasons why code compiled statically, without profile-guided optimization, cannot be as good as JIT compiled code in the extreme case: the lack of profiling data to drive, for instance, inline caching.^[8]

There also exist Java implementations that combine an AOT (ahead-of-time) compiler with either a JIT compiler (Excelsior JET) or interpreter (GNU Compiler for Java.)

History

The earliest published JIT compiler is generally attributed to work on LISP by McCarthy in 1960.^[9] In his seminal paper *Recursive functions of symbolic expressions and their computation by machine, Part I*, he mentions functions that are translated during runtime, thereby sparing the need to save the compiler output to punch cards.^[10] In 1968, Thompson presented a method to automatically compile regular expressions to machine code, which is then executed in order to perform the matching on an input text.^{[9] [11]} An influential technique for deriving compiled code from interpretation was pioneered by Mitchell in 1970, which he implemented for the experimental language *LC²*.^{[9] [12]}

Smalltalk pioneered new aspects of JIT compilations. For example, translation to machine code was done on demand, and the result was cached for later use. When memory became scarce, the system would delete some of this code and regenerate it when it was needed again.^{[9] [13]} Sun's Self language improved these techniques extensively and was at one point the fastest Smalltalk system in the world; achieving up to half the speed of optimized C^[14] but with a fully object-oriented language.

Self was abandoned by Sun, but the research went into the Java language, and currently it is used by most implementations of the Java Virtual Machine, as HotSpot builds on, and extensively uses, this research base.

The HP project Dynamo was an experimental JIT compiler where the 'bytecode' format and the machine code format were the same; the system turned HPA-8000 machine code into HPA-8000 machine code. Counterintuitively, this resulted in speed ups, in some cases of 30% since doing this permitted optimizations at the machine code level, for example, inlining code for better cache usage and optimizations of calls to dynamic libraries and many other run-time optimizations which conventional compilers are not able to attempt.^[15]

References

- Free Online Dictionary of Computing entry [16]
- [1] Kogent Solution Inc. (2007). *Java 6 Programming Black Book, New Ed* (<http://books.google.com/books?id=SSyuJa04uv8C>). Dreamtech Press. p. 5. ISBN 9788177227369. .
- [2] Haase, Chet (May 2007). "Consumer JRE: Leaner, Meaner Java Technology" (<http://java.sun.com/developer/technicalArticles/javase/consumerjre#Quickstarter>). Sun Microsystems. . Retrieved 2007-07-27. "At the OS level, all of these megabytes have to be read from disk, which is a very slow operation. Actually, it's the seek time of the disk that's the killer; reading large files sequentially is relatively fast, but seeking the bits that we actually need is not. So even though we only need a small fraction of the data in these large files for any particular application, the fact that we're seeking all over within the files means that there is plenty of disk activity. "
- [3] The Java HotSpot Performance Engine Architecture (<http://java.sun.com/products/hotspot/whitepaper.html>)
- [4] Schilling, Jonathan L. (February 2003). "The simplest heuristics may be the best in Java JIT compilers". *SIGPLAN Notices* **38** (2): 36–46. doi:10.1145/772970.772975.
- [5] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani, "A dynamic optimization framework for a Java just-in-time compiler", *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01)*, pp. 180–195, October 14–18, 2001.
- [6] Matthew Arnold, Michael Hind, Barbara G. Ryder, "An Empirical Study of Selective Optimization", *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, pp. 49–67, August 10–12, 2000.
- [7] [http://msdn2.microsoft.com/en-us/library/6t9t5wcf\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/6t9t5wcf(VS.80).aspx)
- [8] Matthew R. Arnold, Stephen Fink, David P. Grove, Michael Hind, and Peter F. Sweeney, " A Survey of Adaptive Optimization in Virtual Machines (<http://www.research.ibm.com/people/h/hind/papers.html#survey05>)", *Proceedings of the IEEE*, 92(2), February 2005, pp. 449–466.
- [9] Acock, J. (June 2003). "A brief history of just-in-time" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.3985&rep=rep1&type=pdf>). *ACM Computing Surveys* **35** (2): 97–113. doi:10.1145/857076.857077. . Retrieved 2010-05-24.
- [10] McCarthy, J. (April 1960). "Recursive functions of symbolic expressions and their computation by machine, Part I" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.8833&rep=rep1&type=pdf>). *Communications of the ACM* **3** (4): 184–195. doi:10.1145/367177.367199. . Retrieved 24 May 2010.
- [11] Thompson, K. (June 1968). "Programming Techniques: Regular expression search algorithm" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.9868&rep=rep1&type=pdf>). *Communications of the ACM* **11** (6): 419–422. doi:10.1145/363347.363387. . Retrieved 2010-05-24.
- [12] Mitchell, J.G. (1970). *The design and construction of flexible and efficient interactive programming systems*
- [13] Deutsch, L.P.; Schiffman, A.M. (1984). "Efficient implementation of the Smalltalk-80 system" (<http://webpages.charter.net/allanms/popl84.pdf>). *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*: 297–302. doi:10.1145/800017.800542.
- [14] <http://research.sun.com/jtech/pubs/97-pep.ps>
- [15] Ars Technica on HP's Dynamo (<http://arstechnica.com/reviews/1q00/dynamo/dynamo-1.html>)
- [16] <http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?just-in-time>

External links

- libJIT (<http://freshmeat.net/projects/libjit/>) at Freshmeat — A library by Rhys Weatherley, Klaus Treichel, Aleksey Demakov, and Kirill Kononenko for development of Just-In-Time compilers in Virtual Machine implementations, Dynamic programming languages and Scripting languages.
- Mozilla Nanojit (<https://developer.mozilla.org/En/Nanojit>) — A small, cross-platform C++ library that emits machine code. It is used as the JIT for the Mozilla Tamarin and SpiderMonkey Javascript engines.
- SoftWire (<https://gna.org/projects/softwire/>) — A library by Nicolas Capens that generates assembly language code at run-time (thesis (<http://search.ugent.be/meercat/x/view/rug01/001312059>))
- CCG (http://vvm.lip6.fr/projects_realizations/ccg/) by Ian Piumarta
- Dyninst (<http://www.dyninst.org/>)
- AsmJit (<http://code.google.com/p/asmjit/>) — Complete x86/x64 jit assembler library for C++ language by Petr Kobalíček
- Xbyak (http://homepage1.nifty.com/herumi/soft/xbyak_e.html) — A x86/x64 JIT assembler for C++ language by Herumi
- Profiling Runtime Generated and Interpreted Code using the VTune Performance Analyzer (http://software.intel.com/sites/products/documentation/hpc/vtune/windows/jit_profiling.pdf)

Bytecode

Bytecode, also known as p-code (portable code), is a term which has been used to denote various forms of instruction sets designed for efficient execution by a software interpreter as well as being suitable for further compilation into machine code. Since instructions are processed by software, they may be arbitrarily complex, but are nonetheless often akin to traditional hardware instructions; virtual stack machines are the most common, but virtual register machines have also been built.^[1] Different parts may often be stored in separate files, similar to object modules, but dynamically loaded during execution.

The name bytecode stems from instruction sets which have one-byte opcodes followed by optional parameters. Intermediate representations such as bytecode may be output by programming language implementations to ease interpretation, or it may be used to reduce hardware and operating system dependence by allowing the same code to run on different platforms. Bytecode may often be either directly executed on a virtual machine (i.e. interpreter), or it may be further compiled into machine code for better performance.

Unlike human-readable source code, bytecodes are compact numeric codes, constants, and references (normally numeric addresses) which encode the result of parsing and semantic analysis of things like type, scope, and nesting depths of program objects. They therefore allow much better performance than direct interpretation of source code.

Execution

A bytecode program may be executed by parsing and *directly* executing the instructions, one at a time. This kind of *bytecode interpreter* is very portable. Some systems, called dynamic translators, or "just-in-time" (JIT) compilers, translate bytecode into machine language as necessary at runtime: this makes the virtual machine unportable, but doesn't lose the portability of the bytecode itself. For example, Java and Smalltalk code is typically stored in bytecoded format, which is typically then JIT compiled to translate the bytecode to machine code before execution. This introduces a delay before a program is run, when bytecode is compiled to native machine code, but improves execution speed considerably compared to direct interpretation of the source code—normally by several magnitudes. Because of its performance advantage, today many language implementations execute a program in two phases, first compiling the source code into bytecode, and then passing the bytecode to the virtual machine. Therefore, there are virtual machines for Java, Python, PHP,^[2] Forth, and Tcl. The implementation of Perl and Ruby 1.8 instead work by walking an abstract syntax tree representation derived from the source code.

Examples

- ActionScript executes in the ActionScript Virtual Machine (AVM), which is part of Flash Player and AIR. ActionScript code is typically transformed into bytecode format by a compiler. Examples of compilers include the one built in to Adobe Flash Professional and the one that is built in to Adobe Flash Builder and available in the Adobe Flex SDK.
- Adobe Flash objects
- BANCStar, originally bytecode for an interface-building tool but used as a language in its own right.
- Byte Code Engineering Library
- C to Java Virtual Machine compilers
- CLISP implementation of Common Lisp compiles only to bytecode
- CMUCL and Scieneer Common Lisp implementations of Common Lisp can compile either to bytecode or to native code; bytecode is much more compact
- Dalvik bytecode, designed for the Android platform, is executed by the Dalvik virtual machine.
- EiffelStudio for the Eiffel programming language

- Emacs is a text editor with a majority of its functionality implemented by its specific dialect of Lisp. These features are compiled into bytecode. This architecture allows users to customize the editor with a high level language, which after compilation into bytecode yields reasonable performance.
 - Embeddable Common Lisp implementation of Common Lisp can compile to bytecode or C code
 - Ericsson implementation of Erlang uses BEAM bytecodes
 - Icon^[3] and Unicon^[4] programming languages
 - Infocom used the Z-machine to make its software applications more portable.
 - Java bytecode, which is executed by the Java Virtual Machine
 - ASM
 - BCEL
 - Javassist
 - JMangler
 - LLVM, a modular bytecode compiler and virtual machine
 - Lua, using a register-based virtual machine, also compiles LUAC forms of its scripts for small fast systems that need not include the compiler.
 - m-code of the MATLAB programming language^[5]
 - Managed code such as Microsoft .NET Common Intermediate Language, executed by the .NET Common Language Runtime (CLR)
 - O-code of the BCPL programming language
 - Objective Caml (Ocaml) programming language optionally compiles to a compact bytecode form
 - p-code of UCSD Pascal implementation of the Pascal programming language
 - Parrot virtual machine
 - The R environment for statistical computing offers a byte code compiler through the compiler package, now standard with R version 2.13.0. It is possible to compile this version of R so that the base and recommended packages take advantage of this.^[6]
 - Scheme 48 implementation of Scheme using bytecode interpreter
 - Bytecodes of many implementations of the Smalltalk programming language
 - The SPIN interpreter built into the Parallax Propeller Microcontroller
 - SWEET16
 - Visual FoxPro compiles to bytecode
 - YARV and Rubinius for Ruby.

Notes

Dynamic compilation

Dynamic compilation is a process used by some programming language implementations to gain performance during program execution. Although the technique originated in the Self programming language, the best-known language that uses this technique is Java. Since the machine code emitted by a dynamic compiler is constructed and optimized at program runtime, the use of dynamic compilation enables optimizations for efficiency not available to compiled programs except through code duplication or metaprogramming.

Runtime environments using dynamic compilation typically have programs run slowly for the first few minutes, and then after that, most of the compilation and recompilation is done and it runs quickly. Due to this initial performance lag, dynamic compilation is undesirable in certain cases. In most implementations of dynamic compilation, some optimizations that could be done at the initial compile time are delayed until further compilation at run-time, causing further unnecessary slowdowns. Just-in-time compilation is a form of dynamic compilation.

A closely related technique is **incremental compilation**. An incremental compiler is used in POP-2, POP-11, some versions of Lisp, e.g. Maclisp and at least one version of the ML programming language (Poplog ML). This requires the compiler for the programming language to be part of the runtime system. In consequence, source code can be read in at any time, from the terminal, from a file, or possibly from a data-structure constructed by the running program, and translated into a machine code block or function (which may replace a previous function of the same name), which is then immediately available for use by the program. Because of the need for speed of compilation during interactive development and testing, the compiled code is likely not to be as heavily optimised as code produced by a standard 'batch compiler', which reads in source code and produces object files that can subsequently be linked and run. However an incrementally compiled program will typically run much faster than an interpreted version of the same program. Incremental compilation thus provides a mixture of the benefits of interpreted and compiled languages. To aid portability it is generally desirable for the incremental compiler to operate in two stages, namely first compiling to some intermediate platform-independent language, and then compiling from that to machine code for the host machine. In this case porting requires only changing the 'back end' compiler. Unlike dynamic compilation, as defined above, incremental compilation does not involve further optimisations after the program is first run.

External links

- The UW Dynamic Compilation Project ^[1]
- Architecture Emulation through Dynamic Compilation ^[2]
- SCIRun ^[3]
- Article "Dynamic Compilation, Reflection, & Customizable Apps" ^[4] by David B. Scofield and Eric Bergman-Terrell
- Article "High-performance XML: Dynamic XPath expressions compilation" ^[5] by Daniel Cazzulino
- Matthew R. Arnold, Stephen Fink, David P. Grove, Michael Hind, and Peter F. Sweeney, A Survey of Adaptive Optimization in Virtual Machines ^[6], Proceedings of the IEEE, 92(2), February 2005, Pages 449-466.

References

- [1] <http://www.cs.washington.edu/research/dyncomp/>
- [2] <http://www.research.ibm.com/daisy/>
- [3] <http://software.sci.utah.edu/doc/Developer/Guide/dev.dynamiccomp.html>
- [4] <http://www.ddj.com/documents/ddj0410h/>
- [5] <http://weblogs.asp.net/cazzu/archive/2003/10/07/30888.aspx>
- [6] <http://www.research.ibm.com/people/h/hind/papers.html#survey05>

Dynamic recompilation

In computer science, **dynamic recompilation** (sometimes abbreviated to **dynarec** or the pseudo-acronym **DRC**) is a feature of some emulators and virtual machines, where the system may recompile some part of a program *during execution*. By compiling during execution, the system can tailor the generated code to reflect the program's run-time environment, and perhaps produce more efficient code by exploiting information that is not available to a traditional static compiler.

In other cases, a system may employ dynamic recompilation as part of an adaptive optimization strategy to execute a portable program representation such as Java or .NET Common Language Runtime bytecodes. Full-speed debuggers could also utilize it to reduce the space overhead incurred in most deoptimization techniques, and many other features such as dynamic thread migration.

Example

Suppose a program is being run in an emulator and needs to copy a null-terminated string. The program is compiled originally for a very simple processor. This processor can only copy a byte at a time, and must do so by first reading it from the source string into a register, then writing it from that register into the destination string. The original program might look something like this:

```

beginning:
    mov A,[first string pointer]      ; Put location of first character of source string
                                         ; in register A
    mov B,[second string pointer]     ; Put location of first character of destination string
                                         ; in register B

loop:
    mov C,[A]                      ; Copy byte at address in register A to register C
    mov [B],C                       ; Copy byte in register C to the address in register B
    inc A                          ; Increment the address in register A to point to
                                     ; the next byte
    inc B                          ; Increment the address in register B to point to
                                     ; the next byte
    cmp C,#0                       ; Compare the data we just copied to 0 (string end marker)
    jnz loop                       ; If it wasn't 0 then we have more to copy, so go back
                                     ; and copy the next byte
end:                           ; If we didn't loop then we must have finished,
                                     ; so carry on with something else.

```

The emulator might be running on a processor which is similar, but extremely good at copying strings, and the emulator knows it can take advantage of this. It might recognize the string copy sequence of instructions and decide to rewrite them more efficiently just before execution, to speed up the emulation.

Say there is an instruction on our new processor called *movs*, specifically designed to copy strings efficiently. Our theoretical *movs* instruction copies 16 bytes at a time, without having to load them into register C in between, but will stop if it copies a 0 byte (which marks the end of a string) and set the zero flag. It also knows that the addresses of the strings will be in registers A and B, so it increments A and B by 16 every time it executes, ready for the next copy.

Our new recompiled code might look something like this:

```
beginning:
    mov A, [first string pointer]      ; Put location of first character of source string
                                         ; in register A
    mov B, [second string pointer]    ; Put location of first character of destination string
                                         ; in register B

loop:
    movs [B], [A]                   ; Copy 16 bytes at address in register A to address
                                         ; in register B, then increment A and B by 16
    jnz loop                      ; If the zero flag isn't set then we haven't reached
                                         ; the end of the string, so go back and copy some more.

end:                           ; If we didn't loop then we must have finished,
                                         ; so carry on with something else.
```

There is an immediate speed benefit simply because the processor doesn't have to load so many instructions to do the same task, but also because the *movs* instruction is likely to be optimized by the processor designer to be more efficient than the sequence used in the first example (for example it may make better use of parallel execution in the processor to increment A and B while it is still copying bytes).

Applications using dynamic recompilation

- Many Java virtual machines feature dynamic recompilation.
- MAME uses dynamic recompilation in its CPU emulators for MIPS, SuperH, PowerPC and even the Voodoo graphics processing units.
- 1964, a Nintendo 64 emulator for x86 hardware.
- Apple's Rosetta for Mac OS X on x86, allows PowerPC code to be run on the x86 architecture.
- Later versions of the Mac 68K emulator used in Mac OS to run 680x0 code on the PowerPC hardware.
- Psyco, a specializing compiler for Python.
- The HP Dynamo project, an example of a transparent binary dynamic optimizer.
- The Vx32 virtual machine employs dynamic recompilation to create OS-independent x86 architecture sandboxes for safe application plugins.
- Microsoft Virtual PC for Mac, used to run x86 code on PowerPC.
- NullDC, a Sega Dreamcast emulator for x86.
- QEMU, an open-source full system emulator.
- The backwards compatibility functionality of the Xbox 360 (i.e. running games written for the original Xbox) is widely assumed to use dynamic recompilation.
- PSEmu Pro, a Sony PlayStation emulator.
- Ultrahle, the first Nintendo 64 emulator to fully run commercial games.
- PCSX2 [1], a Sony PlayStation 2 emulator, has a recompiler called "microVU", the successor of "SuperVU".
- Dolphin, a Nintendo GameCube and Wii emulator, has a dynarec option.
- GCemu^[2], a Nintendo GameCube emulator.
- OVPsim [3], a freely-available full system emulator.
- VirtualBox uses dynamic recompilation

- Valgrind, a programming tool for memory debugging, memory leak detection, and profiling, uses dynamic recompilation.
- Wii64, a Nintendo 64 emulator for the Wii.
- WiiSX, a SONY PlayStation emulator for the Nintendo Wii.
- Mupen64Plus, a multi-platform Nintendo 64 emulator.[4]
- Yabause, a multi-platform Saturn emulator.[5]

External links

- [6] HP Labs' technical report on Dynamo
- [7] Dynamic recompiler tutorial

References

- [1] <http://www.pcsx2.net>
- [2] <http://sourceforge.net/projects/gcemu-project>
- [3] <http://www.ovpworld.org>
- [4] http://pandorawiki.org/Mupen64plus_dynamic_recompiler
- [5] http://wiki.yabause.org/index.php5?title=SH2_dynamic_recompiler
- [6] <http://www.hpl.hp.com/techreports/1999/HPL-1999-77.html>
- [7] <http://web.archive.org/web/20051018182930/www.zenogais.net/Projects/Tutorials/Dynamic+Recompiler.html>

Object file

In computer science, an **object file** is an organized collection of separate, named sequences of machine code. Each sequence, or **object**, typically contains instructions for the host machine to accomplish some task, possibly accompanied by related data and metadata (e.g. relocation information, stack unwinding information, comments, program symbols, debugging or profiling information). A linker is typically used to generate an executable or library by combining parts of object files.

Object file formats

An **object file format** is a computer file format used for the storage of object code and related data typically produced by a compiler or assembler.

There are many different object file formats; originally each type of computer had its own unique format, but with the advent of Unix and other portable operating systems, some formats, such as COFF and ELF, have been defined and used on different kinds of systems. It is common for the same file format to be used both as linker input and output, and thus as the library and **executable file format**.

The design and/or choice of an object file format is a key part of overall system design. It affects the performance of the linker and thus programmer turnaround while developing. If the format is used for executables, the design also affects the time programs take to begin running, and thus the responsiveness for users. Most object file formats are structured as blocks of data, each block containing a certain type of data (see Memory segmentation). These blocks can be paged in as needed by the virtual memory system, needing no further processing to be ready to use.

One simple object file format is the DOS .COM format, which is simply a file of raw bytes that is always loaded at a fixed location. Other formats are an elaborate array of structures and substructures whose specification runs to many pages.

Debugging information may either be an integral part of the object file format, as in COFF, or a semi-independent format which may be used with several object formats, such as stabs or DWARF.

The GNU Project's Binary File Descriptor library (BFD library) provides a common API for the manipulation of object files in a variety of formats.

Types of data supported by typical object file formats:

- BSS (Block Started by Symbol)
- Text segment
- Data segment

Code segment

In computing, a **code segment**, also known as a **text segment** or simply as **text**, is one of the sections of a program in an object file or in memory, which contains executable instructions.

It has a fixed size and is usually read-only. If the text section is not read-only, then the particular architecture allows self-modifying code. Fixed-position or position independent code may be shared in memory by several processes in segmented or paged memory systems.

As a memory region, a **code segment** may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

External links

- mem_sequence.c - A program that sequentially lists memory regions in itself^[1]

References

[1] <http://blog.ooz.ie/2008/09/0x03-notes-on-assembly-memory-from.html>

Data segment

A **data segment** is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. This portion has a fixed size for each program depending upon the quantity of contents, since all of the data here is set by the programmer before the program is loaded.

Note that, **data segment** is not read-only, since the values of the variables can be altered at run time. This is in contrast to the Rodata (constant, read-only data) section, as well as the code segment (also known as text segment).

The PC architecture supports few basic read-write memory regions in a program namely: Stack, Data and Code. A Heap is another region of address space of a program which is managed by operating system in terms how the program uses it of which it is a part of.

Program memory

The computer program memory is organized into the following:

- Data Segment (Data + BSS + Heap)
- Stack
- Code segment

Data

The data area contains global and static variables used by the program that are initialized. This segment can be further classified into initialized read-only area and initialized read-write area. For instance the string defined by `char s[] = "hello world"` in C and a C statement like `int debug=1` outside the "main" would be stored in initialized read-write area. And a C statement like `const char* string = "hello world"` makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable `string` in initialized read-write area. Ex: `static int i = 10` will be stored in data segment **and** `global int i = 10` will be stored in data segment

BSS

The **BSS segment** also colloquially known as *uninitialized data* starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. For instance a variable declared `static int i;` would be contained in the BSS segment.

Heap

The heap area begins at the end of the **BSS segment** and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single "heap area" is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process' virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Stack

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program **stack**, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame"; A stack frame consists at minimum of a return address.

References

- "BraveGNU.org" [1].

External links

- mem_sequence.c - sequentially lists memory regions in a process [1]
- Expert C Programming: Deep C Secrets, Peter van der Linden, Prentice Hall 1997, p. 119ff [2]
- Understanding Memory [3]

References

- [1] <http://www.bravegnu.org/gnu-eprog/c-startup.html>
- [2] <http://www.comp.nus.edu.sg/~xujia/Expert.C.Programming.pdf>
- [3] <http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html>

.bss

In computer programming, the name **.bss** or **bss** is used by many compilers and linkers for a part of the data segment containing statically-allocated variables represented solely by zero-valued bits initially (i.e., when execution begins). It is often referred to as the "bss section" or "bss segment".

In C, statically-allocated variables without an explicit initializer are initialized to zero (for arithmetic types) or a null pointer (for pointer types). Implementations of C typically represent zero values and null pointer values using a bit pattern consisting solely of zero-valued bits (though this is not required by the C standard). Hence, the bss section typically includes all uninitialized variables declared at the file level (i.e., outside of any function) as well as uninitialized local variables declared with the `static` keyword. An implementation may also assign statically-allocated variables initialized with a value consisting solely of zero-valued bits to the bss section.

Typically, the program loader initializes the memory allocated for the bss section when it loads the program. Operating systems may use a technique called zero-fill-on-demand to efficiently implement the bss segment (McKusick & Karels 1986). In embedded software, the bss segment is mapped into memory that is initialized to zero by the C run-time system before `main()` is entered.

Some application binary interfaces also support an **sbss** segment for "small data". Typically, these data items can be accessed by leaner code using instructions that can only access a certain range of addresses.

Historically, **BSS** (from **Block Started by Symbol**) was a pseudo-operation in UA-SAP (United Aircraft Symbolic Assembly Program), the assembler developed in the mid-1950s for the IBM 704 by Roy Nutt, Walter Ramshaw, and others at United Aircraft Corporation [1] [2]. The BSS keyword was later incorporated into FAP (FORTRAN Assembly Program), IBM's standard assembler for its 709 and 7090/94 computers. It defined a label (i.e. symbol) and reserved a block of uninitialized space for a given number of words (Timar 1996).

A noted C programmer and author says "Some people like to remember it as "Better Save Space." Since the BSS segment only holds variables that don't have any value yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but BSS (unlike the data segment) doesn't take up any actual space in the object file."^[3]

References

- Stevens, W. Richard (1992). *Advanced Programming in the Unix Environment*. Addison-Wesley. Section 7.6. ISBN 0-201-56317-7.
- Timar, Ted; et al. (1996). "Unix - Frequently Asked Questions (1/7)"^[4]. Question 1.3.
- McKusick, Marshall Kirk; Karels, Michael J. (1986). "A New Virtual Memory Implementation for Berkeley UNIX"^[5]. University of California, Berkeley. p. 3.

References

- [1] *Network Dictionary*. Javvin Press, 2007, p. 70.
- [2] Coding for the MIT-IBM 704 Computer October 1957, p. V-10 (http://bitsavers.org/pdf/mit/computer_center/Coding_for_the/MIT-IBM_704_Computer_Oct57.pdf)
- [3] Expert C Programming: Deep C Secrets, Peter van der Linden, Prentice Hall 1997, p. 123 (<http://www.comp.nus.edu.sg/~xujia/Expert.C.Programming.pdf>)
- [4] <http://www.faqs.org/faqs/unix-faq/faq/part1/section-3.html>
- [5] <http://docs.freebsd.org/44doc/papers/newvm.pdf>

Literal pool

In computer science, and specifically in compiler and assembler design, a **literal pool** is a lookup table used to hold literals during assembly and execution.

Multiple (local) literal pools are typically used only for computer architectures that lack branch instructions for long jumps, or have a set of instructions optimized for shorter jumps. Examples of such architectures include ARM architecture and the IBM System/360 and later architectures, which had a number of instructions which took 12-bit address offsets. In this case, the compiler would create a literal table on every 4K page; any branches whose target was less than 4K bytes away could be taken immediately; longer branches required an address lookup via the literal table. The entries in the literal pool are placed into the object relocation table during assembly, and are then resolved at link edit time.

In certain ways, a literal pool resembles a TOC or a global offset table (GOT), except that the implementation is considerably simpler, and there may be multiple literal tables per object.

Overhead code

Overhead code is the additional (or excess) object code generated by a compiler to provide machine code which will be executed by a specific CPU. This code includes translations of generic instructions listed in cross-platform code, and is tailored for execution on a specific platform or architecture. An example of overhead code would be code generated to handle reference counting, while source code written in a high level language is an example cross-platform code.

Analogy

The coverage area of a standalone house is more than the space taken by enclosures. Similarly, overhead code is the part of the program which is not listed in the source code. However, it is needed for the software to operate properly.

Link time

In computer science, **link time** refers to either the operations performed by a linker (ie, *link time operations*) or programming language requirements that must be met by compiled source code for it to be successfully linked (ie, *link time requirements*).

The operations performed at link time usually include fixing up the addresses of externally referenced objects and functions, various kinds of cross module checks (eg, type checks on externally visible identifiers and in some languages instantiation of template). Some optimizing compilers delay code generation until link time because it is here that information about a complete program is available to them.

The definition of a programming language may specify link time requirements that source code must meet to be successfully compiled (eg, the maximum number of characters in an externally visible identifier that must be considered significant).

Link time occurs after compile time and before runtime (when a program is executed). In some programming languages it may be necessary for some compilation and linking to occur at runtime.

Relocation

In computer science, **relocation** is the process of replacing symbolic references or names of libraries with actual usable addresses in memory before running a program. It is typically done by the linker during compilation (at compile time), although it can be done at runtime by a relocating loader. Compilers or assemblers typically generate the executable with zero as the lower-most starting address. Before the execution of object code, these addresses should be adjusted so that they denote the correct runtime addresses.

Relocation is typically done in two steps:

1. Each object code has various sections like code, data, .bss etc. To combine all the objects to a single executable, the linker merges all sections of similar type into a single section of that type. The linker then assigns runtime addresses to each section and each symbol. At this point, the code (functions) and data (global variables) will have unique runtime addresses.
2. Each section refers to one or more symbols which should be modified so that they point to the correct runtime addresses.

A **fixup table** -- a relocation table -- can also be provided in the header of the object code file. Each "fixup" is a pointer to an address in the object code that must be changed when the loader relocates the program. Fixups are designed to support relocation of the program as a complete unit. In some cases, each fixup in the table is itself relative to a base address of zero, so the fixups themselves must be changed as the loader moves through the table.^[1] In some architectures, compilers, and executable models, a fixup that crosses certain boundaries (such as a segment boundary) or that does not lie on a word boundary is illegal and flagged as an error by the linker.^[2]

References

[1] John R. Levine (October 1999). "Chapter 3: Object Files". *Linkers and Loaders*. Morgan-Kauffman. ISBN 1-55860-496-0.

[2] "Borland article #15961: Coping with 'Fixup Overflow' messages" (<http://web.archive.org/web/20070324041806/http://vmlinux.org/~jakov/community.borland.com/15961.html>). Archived from the original (<http://vmlinux.org/~jakov/community.borland.com/15961.html>) on 2007-03-24. Retrieved 2007-01-15.

Library

In computer science, a **library** is a collection of resources used to develop software. These may include pre-written code and subroutines, classes, values or type specifications.

Libraries contain code and data that provide services to independent programs. This encourages the sharing and changing of code and data in a modular fashion, and eases its distribution. Some executables are both standalone programs and libraries, but most libraries are not executable. Executables and libraries make references known as *links* to each other through the process known as *linking*, which is typically done by a linker.

Most compiled languages have a standard library although programmers can also create their own custom libraries. Most modern software systems of 2009 provide libraries that implement the majority of system services. Such libraries have commoditized the services which a modern application requires. As such, most code used by modern applications is provided in these system libraries.

The GPL linking exception allows programs which do not license themselves under GPL to link to libraries licensed under the exception without thereby becoming subject to GPL requirements.

Libraries often contain a jump table of all the methods within it, known as *entry points*. Calls into the library use this table, looking up the location of the code in memory, then calling it. This introduces overhead in calling into the library, but the delay is so small as to be negligible.

History

The earliest programming concepts analogous to libraries were intended to separate data definitions from the program implementation. JOVIAL brought the "COMPOOL" (Communication Pool) concept to popular attention in 1959, although it adopted the idea from the large-system SAGE software. Following the computer science principles of separation of concerns and information hiding, "Comm Pool's purpose was to permit the sharing of System Data among many programs by providing a centralized data description."^[1]

COBOL also included "primitive capabilities for a library system" in 1959,^[2] but Jean Sammet described them as "inadequate library facilities" in retrospect.^[3]

Another major contributor to the modern library concept came in the form of the subprogram innovation of FORTRAN. FORTRAN subprograms can be compiled independently of each other, but the compiler lacks a linker. So prior to the introduction of modules in Fortran-90, type checking between subprograms was impossible.^[4]

Finally, historians of the concept should remember the influential Simula 67. Simula was the first object-oriented programming language, and its classes are nearly identical to the modern concept as used in Java, C++, and C#. The *class* concept of Simula was also a progenitor of the *package* in Ada and the *module* of Modula-2.^[5] Even when developed originally in 1965, Simula classes could be included in library files and added at compile time.^[6]

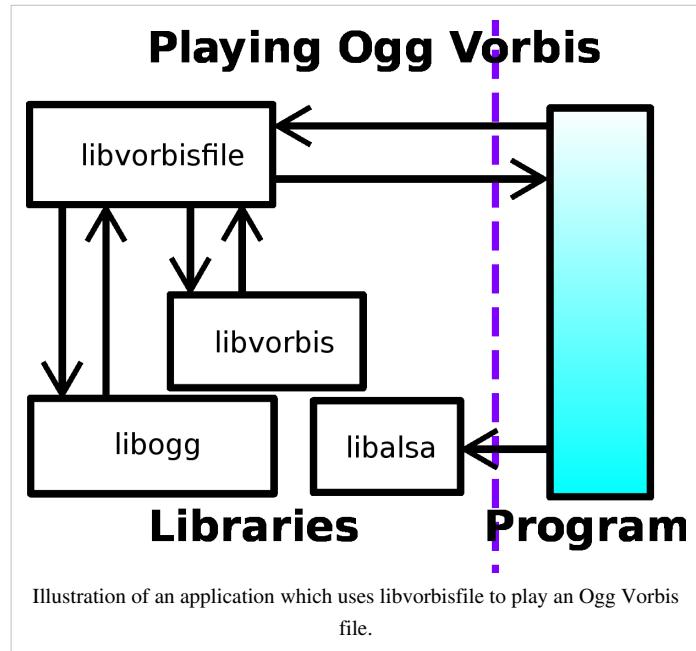


Illustration of an application which uses libvorbisfile to play an Ogg Vorbis file.

Linking

Libraries are important in the program *linking* or *binding* process, which resolves references known as *links* or *symbols* to library modules. The linking process is usually automatically done by a *linker* or *binder* program that searches a set of libraries and other modules in a given order. Usually it is not considered an error if a link target can be found multiple times in a given set of libraries. Linking may be done when an executable file is built during compile time or link time, or whenever the program is used at run time.

The references being resolved may be addresses for jumps and other routine calls. They may be in the main program, or in one module depending upon another. They are resolved into fixed or relocatable addresses (from a common base) by allocating runtime memory for the memory segments of each module referenced.

Some programming languages may use a feature called *smart linking* wherein the linker is aware of or integrated with the compiler, such that the linker knows how external references are used, and code in a library that is never actually *used*, even though internally referenced, can be discarded from the compiled application. For example, a program that only uses integers for arithmetic, or does no arithmetic operations at all, can exclude floating-point library routines. This smart-linking feature can lead to smaller application file sizes and reduced memory usage.

Relocation

Some references in a program or library module are stored in a relative or symbolic form which cannot be resolved until all code and libraries are assigned final static addresses. *Relocation* is the process of adjusting these references, and is done either by the linker or the loader. In general, relocation cannot be done to individual libraries themselves because the addresses in memory may vary depending on the program using them and other libraries they are combined with. Position-independent code avoids references to absolute addresses and therefore does not require relocation.

Static libraries

When linking is done once, when an executable or other object file is created, it is known as *static linking* or *early binding*. In this case the linking may be done by the compiler. A *static library*, also known as an *archive*, is one intended to be statically linked. Originally, only static libraries existed. Static linking must be performed when any modules are recompiled.

All of the modules required by a program are sometimes statically linked and copied into the executable file. This process, and the resulting stand-alone file, is known as a static build of the program. A static build may not need any further relocation if virtual memory is used and no address space layout randomization is desired.

Shared libraries

A **shared library** or **shared object** is a file that is intended to be shared by executable files and further shared objects files. Modules used by a program are loaded from individual shared objects into memory at load time or run-time, rather than being copied into a single monolithic file by the linker at compile time.

Shared libraries can be statically linked, meaning that references to the library modules are resolved and the modules are allocated memory at compile time. But often linking of shared libraries is postponed until they are loaded.

Most modern operating systems can have shared library files of the same format as the executable files. This offers two main advantages: first, it requires making only one loader for both of them, rather than two (having the single loader is considered well worth its added complexity). Secondly, it allows the executables also to be used as shared libraries, if they have a symbol table. Typical combined executable and shared library formats are ELF and Mach-O (both in Unix) and PE (Windows). In Windows, the concept was taken one step further, with even system resources such as fonts being bundled in the PE format. The same is true under OpenStep, where the universal "bundle" format

is used for almost all system resources.

Memory sharing

Library code may be shared in memory by multiple processes as well as on disk. If virtual memory is used, processes execute the same physical page of RAM, mapped into the different address spaces of each process. This has advantages. For instance on the OpenStep system, applications were often only a few hundred kilobytes in size and loaded quickly; the majority of their code was located in libraries that had already been loaded for other purposes by the operating system. There is a cost, however; shared code must be specifically written to run in a multitasking environment. In some older environments such as 16-bit Windows or MPE for the HP 3000, only stack based data (local) was allowed, or other significant restrictions were placed on writing a shared library.

Programs can accomplish RAM sharing by using position independent code as in Unix, which leads to a complex but flexible architecture, or by using common virtual addresses as in Windows and OS/2. These systems make sure, by various tricks like pre-mapping the address space and reserving slots for each shared library, that code has a great probability of being shared. A third alternative is single-level store, as used by the IBM System/38 and its successors. This allows position-dependent code but places no significant restrictions on where code can be placed or how it can be shared.

In some cases different versions of shared libraries can cause problems, especially when libraries of different versions have the same file name, and different applications installed on a system each require a specific version. Such a scenario is known as DLL hell, named after the Windows and OS/2 DLL file. Most modern operating systems after 2001, have clean-up methods to eliminate such situations.

Dynamic linking

Dynamic linking or **late binding** refers to linking performed every time a program is executed, at load time or run-time, rather than before the executable file is created at compile time or link time. A **dynamically linked library** (**dynamic-link library** or **DLL** under Windows and OS/2; **dynamic shared object** or **DSO** under Unix-like systems) is a library intended for dynamic linking. Only a minimum amount of work is done at compile time or link time by the linker; it only records what library routines the program needs and the index names or numbers of the routines in the library. The majority of the work of linking is done at the time the application is loaded (loadtime) or during execution (runtime). The necessary linking code, called a *dynamic linker* or *linking loader*, is actually part of the underlying operating system.

Programmers originally developed dynamic linking in the Multics operating system, starting in 1964, and the MTS (Michigan Terminal System), built in the late 1960s.^[7]

Optimizations

Since shared libraries on most systems do not change often, systems can compute a likely load address for each shared library on the system before it is needed, and store that information in the libraries and executables. If every shared library that is loaded has undergone this process, then each will load at its predetermined address, which speeds up the process of dynamic linking. This optimization is known as prebinding in Mac OS X and prelinking in Linux. Disadvantages of this technique include the time required to precompute these addresses every time the shared libraries change, the inability to use address space layout randomization, and the requirement of sufficient virtual address space for use (a problem that will be alleviated by the adoption of 64-bit architectures, at least for the time being).

Locating libraries at runtime

Loaders for shared libraries vary widely in functionality. Some depend on the executable storing explicit paths to the libraries. Any change to the library naming or layout of the file system will cause these systems to fail. More commonly, only the name of the library (and not the path) is stored in the executable, with the operating system supplying a method to find the library on-disk based on some algorithm.

If a shared library that an executable depends on is deleted, moved, or renamed, or if an incompatible version of the library is copied to a place that is earlier in the search, the executable would fail to load. On Windows this is commonly known as DLL hell.

AmigaOS

AmigaOS stores generic system libraries in a directory defined by the *LIBS*: path assignment. Application-specific libraries can go in the same directory as the application's executable. AmigaOS will search these locations when an executable attempts to launch a shared library. An application may also supply an explicit path when attempting to launch a library.

Microsoft Windows

Microsoft Windows will check the registry to determine the proper place to find an ActiveX DLL, but for other DLLs it will check the directory from where it loaded the program; the current working directory; any directories set by calling the `SetDllDirectory()` function; the System32, System, and Windows directories; and finally the directories specified by the PATH environment variable.^[8] Applications written for the .NET Framework framework (since 2002), also check the Global Assembly Cache as the primary store of shared dll files to remove the issue of DLL hell.

OpenStep

OpenStep used a more flexible system, collecting a list of libraries from a number of known locations (similar to the PATH concept) when the system first starts. Moving libraries around causes no problems at all, although users incur a time cost when first starting the system.

Unix-like systems

Most Unix-like systems have a "search path" specifying file system directories in which to look for dynamic libraries. Some systems specify the default path in a configuration file; others hard-code it into the dynamic loader. Some executable file formats can specify additional directories in which to search for libraries for a particular program. This can usually be overridden with an environment variable, although it is disabled for setuid and setgid programs, so that a user can't force such a program to run arbitrary code with root permissions. Developers of libraries are encouraged to place their dynamic libraries in places in the default search path. On the downside, this can make installation of new libraries problematic, and these "known" locations quickly become home to an increasing number of library files, making management more complex.

Dynamic loading

Dynamic loading, a subset of dynamic linking, involves a dynamically linked library loading and unloading at run-time on request. Such a request may be made implicitly at compile-time or explicitly at run-time. Implicit requests are made at compile-time when a linker adds library references that include file paths or simply file names. Explicit requests are made when applications make direct calls to an operating system's API at runtime.

Most operating systems that support dynamically linked libraries also support dynamically loading such libraries via a run-time linker API. For instance, Microsoft Windows uses the API functions `LoadLibrary`, `LoadLibraryEx`, `FreeLibrary` and `GetProcAddress` with Microsoft Dynamic Link Libraries; POSIX based systems, including most UNIX and UNIX-like systems, use `dlopen`, `dlclose` and `dlsym`. Some development systems automate this process.

Object and class libraries

Although originally pioneered in the 1960s, dynamic linking did not reach operating systems used by consumers until the late 1980s. It was generally available in some form in most operating systems by the early 1990s. During this same period, object-oriented programming (OOP) was becoming a significant part of the programming landscape. OOP with runtime binding requires additional information that traditional libraries don't supply. In addition to the names and entry points of the code located within, they also require a list of the objects on which they depend. This is a side-effect of one of OOP's main advantages, inheritance, which means that the complete definition of any method may be defined in a number of places. This is more than simply listing that one library requires the services of another: in a true OOP system, the libraries themselves may not be known at compile time, and vary from system to system.

At the same time many developers worked on the idea of multi-tier programs, in which a "display" running on a desktop computer would use the services of a mainframe or minicomputer for data storage or processing. For instance, a program on a GUI-based computer would send messages to a minicomputer to return small samples of a huge dataset for display. Remote procedure calls already handled these tasks, but there was no standard RPC system.

Soon the majority of the minicomputer and mainframe vendors instigated projects to combine the two, producing an OOP library format that could be used anywhere. Such systems were known as **object libraries**, or **distributed objects**, if they supported remote access (not all did). Microsoft's COM is an example of such a system for local use, DCOM a modified version that supports remote access.

For some time object libraries held the status of the "next big thing" in the programming world. There were a number of efforts to create systems that would run across platforms, and companies competed to try to get developers locked into their own system. Examples include IBM's System Object Model (SOM/DSOM), Sun Microsystems' Distributed Objects Everywhere (DOE), NeXT's Portable Distributed Objects (PDO), Digital's ObjectBroker, Microsoft's Component Object Model (COM/DCOM), and any number of CORBA-based systems.

After the inevitable cooling of marketing hype, object libraries continue to be used in both object-oriented programming and distributed information systems. **Class libraries** are the rough OOP equivalent of older types of code libraries. They contain classes, which describe characteristics and define actions (methods) that involve objects. Class libraries are used to create instances, or objects with their characteristics set to specific values. In some OOP languages, like Java, the distinction is clear, with the classes often contained in library files (like Java's JAR file format) and the instantiated objects residing only in memory (although potentially able to be made persistent in separate files). In others, like Smalltalk, the class libraries are merely the starting point for a system image that includes the entire state of the environment, classes and all instantiated objects.

Remote libraries

Another solution to the library issue comes from using completely separate executables (often in some lightweight form) and calling them using a remote procedure call (RPC) over a network to another computer. This approach maximizes operating system re-use: the code needed to support the library is the same code being used to provide application support and security for every other program. Additionally, such systems do not require the library to exist on the same machine, but can forward the requests over the network.

However, such an approach means that every library call requires a considerable amount of overhead. RPC calls are much more expensive than calling a shared library that has already been loaded on the same machine. This approach is commonly used in a distributed architecture that makes heavy use of such remote calls, notably client-server systems and application servers such as Enterprise JavaBeans.

Code generation libraries

Code generation libraries are high-level APIs that can generate or transform byte code for Java. They are used by aspect-oriented programming, some data access frameworks, and for testing to generate dynamic proxy objects. They also are used to intercept field access.^[9]

File naming

- AmigaOS

The system identifies libraries with a ".library" suffix (for example: mathieeedoubtrans.library). These are separate files (shared libraries) that are invoked by the program that is running, and are dynamically-loaded but are not dynamically linked. Once the library has been invoked usually it could not deallocate memory and resources it asked for. Users can force a "flushlibs" option by using AmigaDOS command **Avail** (that checks free memory in the system and has the option **Avail flush** that frees memory from libraries left open by programs). Since AmigaOS 4.0 July 2007 First Update, support for shared objects and dynamic linking has been introduced. Now ".so" objects can exist also on Amiga, together with ".library files".

- Most modern Unix-like systems

The system stores libfoo.a and libfoo.so files in directories such as /lib, /usr/lib or /usr/local/lib. The filenames always start with lib, and end with .a (archive, static library) or .so (shared object, dynamically linked library). Some systems might have multiple names for the dynamically linked library, with most of the names being names for symbolic links to the remaining name; those names might include the major version of the library, or the full version number; for example, on some systems libfoo.so.2 would be the filename for the second major interface revision of the dynamically linked library libfoo. The .la files sometimes found in the library directories are libtool archives, not usable by the system as such.

- Mac OS X

The system inherits static library conventions from BSD, with the library stored in a .a file, and can use .so-style dynamically-linked libraries (with the .dylib suffix instead). Most libraries in Mac OS X, however, consist of "frameworks", placed inside special directories called "bundles" which wrap the library's required files and metadata. For example, a framework called MyFramework would be implemented in a bundle called MyFramework.framework, with MyFramework.framework/MyFramework being either the dynamically linked library file or being a symlink to the dynamically linked library file in MyFramework.framework/Versions/Current/MyFramework.

- Microsoft Windows

Dynamically linkable libraries usually have the suffix *.DLL, although other file name extensions may be used for specific purpose dynamically-linked libraries, e.g. *.OCX for OLE libraries. The interface revisions are either encoded in the file names, or abstracted away using COM-object interfaces. Depending on how they are compiled, *.LIB files can be either static libraries or representations of dynamically linkable libraries needed only during compilation, known as "Import Libraries". Unlike in the UNIX world, where different file extensions are used, when linking against .LIB file in Windows one must first know if it is a regular static library or an import library. In the latter case, a .DLL file must be present at runtime.

References

- [1] Wexelblat, Richard (1981). *History of Programming Languages*. ACM Monograph Series. New York, NY: Academic Press (A subsidiary of Harcourt Brace). p. 369. ISBN 0-12-745040-8
- [2] Wexelblat, *op. cit.*, p. 274
- [3] Wexelblat, *op. cit.*, p. 258
- [4] Wilson, Leslie B.; Clark, Robert G. (1988). *Comparative Programming Languages*. Wokingham, England: Addison-Wesley. p. 126. ISBN 0-201-18483-4
- [5] Wilson and Clark, *op. cit.*, p. 52
- [6] Wexelblat, *op. cit.*, p. 716
- [7] "A History of MTS". *Information Technology Digest* 5 (5).
- [8] "Dynamic-Link Library Search Order" (<http://msdn2.microsoft.com/en-us/library/ms682586.aspx>). *Microsoft Developer Network Library*. Microsoft. 2007-10-04. . Retrieved 2007-10-04.
- [9] "Code Generation Library" (<http://sourceforge.net/projects/cglib/>). <http://sourceforge.net/>: Source Forge. . Retrieved 2010-03-03. "Byte Code Generation Library is high level API to generate and transform JAVA byte code. It is used by AOP, testing, data access frameworks to generate dynamic proxy objects and intercept field access."

External links

- Program Library HOWTO for Linux (<http://www.dwheeler.com/program-library/>)
- Shared Libraries - 'Linkers and Loaders' by John R. Levine (<http://www.iecc.com/linker/linker09.html>)
- Dynamic Linking and Loading - 'Linkers and Loaders' by John R. Levine (<http://www.iecc.com/linker/linker10.html>)
- Article *Beginner's Guide to Linkers* (<http://www.lurklurk.org/linkers/linkers.html>) by David Drysdale
- Article *Faster C++ program startups by improving runtime linking efficiency* (<http://objprelink.sourceforge.net/objprelink.html>) by Léon Bottou and John Ryland
- How to Create Program Libraries (<http://www.enderunix.org/simsek/articles/libraries.pdf>) by Baris Simsek
- LIB BFD (http://www.csa.iisc.ernet.in/resources/documentation/hypertext/bfd/bfd_toc.html) - the Binary File Descriptor Library
- 1st Library-Centric Software Design Workshop LCSD'05 at OOPSLA'05 (<http://lcisd05.cs.tamu.edu>)
- 2nd Library-Centric Software Design Workshop LCSD'06 at OOPSLA'06 (<http://lcsd.cs.tamu.edu/2006/>)
- How to create shared library(with much background info) (<http://people.redhat.com/drepper/dsowhowto.pdf>)
- Anatomy of Linux dynamic libraries (<http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/>)
- Criticisms of dynamic linking (<http://harmful.cat-v.org/software/dynamic-linking/>).
- Reusable Software Assets (<http://wfrakes.wordpress.com/category/reusable-assets/>)

Static build

A **static build** is a compiled version of a program which has been **statically** linked against libraries.

In computer science, linking means taking one or more objects generated by compilers and assemble them into a single executable program. The objects are program modules containing machine code and *symbol* definitions, which come in two varieties:

- *Defined* or *exported* symbols are functions or variables that are present in the module represented by the object, and which should be available for use by other modules.
- *Undefined* or *imported* symbols are functions or variables that are called or referenced by this object, but not internally defined.

A linker program then resolves references to undefined symbols by finding out which other object defines a symbol in question, and replacing placeholders with the symbol's address. Linkers can take objects from a collection called a *library*. The final program does not include the whole library, only those objects from it that are needed. Libraries for diverse purposes exist, and one or more system libraries are usually linked in by default.

Modern operating system environments allow *dynamic linking*, or the postponing of the resolving of some undefined symbols until a program is run. That means that the executable still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these. Loading the program will load these objects/libraries as well, and perform a final linking.

In a **statically built** program, no dynamic linking occurs: all the bindings have been done at compile time.

Dynamic linking offers two advantages:

- Often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary.
- If a library is upgraded or replaced, all programs using it dynamically will immediately benefit from the corrections. Static builds would have to be re-linked first.

On the other hand, static builds have a very predictable behavior (because they do not rely on the particular version of libraries available on the final system), and are commonly found in forensic and security tools to avoid possible contamination or malfunction due to broken libraries on the examined machine. The same flexibility that permits an upgraded library to benefit all dynamically-linked applications can also prevent applications that assume the presence of a specific version of a particular library from running correctly. If every application on a system must have its own copy of a dynamic library to ensure correct operation, the benefits of dynamic linking are moot.

Another benefit of static builds is their portability: once the final executable file has been compiled, it is no longer necessary to keep the library files that the program references, since all the relevant parts are copied into the executable file. As a result, when installing a statically-built program on a computer, the user doesn't have to download and install additional libraries: the program is ready to run.

References

- Levine, John R. (2000). *Linkers and Loaders*. San Francisco: Morgan Kaufmann. ISBN 1-55860-496-0.
- Keren, Guy (2002). "Building And Using Static And Shared "C" Libraries" [1]. Little Unix Programmers Group. Retrieved 2007-11-17.

References

[1] <http://users.actcom.co.il/~choo/lug/tutorials/libraries/unix-c-libraries.html>

Architecture Neutral Distribution Format

The **Architecture Neutral Distribution Format (ANDF)** is a technology allowing common "shrink wrapped" binary application programs to be distributed for use on conformant Unix systems, each of which might run on different underlying hardware platforms. ANDF was defined by the Open Software Foundation and was expected to be a "truly revolutionary technology that will significantly advance the cause of portability and open systems",^[1] but it was never widely adopted.

As with other OSF offerings, ANDF was specified through an open selection process. OSF issued a Request for Technology for architecture-neutral software distribution technologies in April, 1989.^[2] Fifteen proposals were received, based on a variety of technical approaches, including obscured source code, compiler intermediate languages, and annotated executable code.

The technology of ANDF, chosen after an evaluation of competing approaches and implementations, was Ten15 Distribution Format, later renamed TenDRA Distribution Format, developed by the UK Defence Research Agency.

Adoption

ANDF was intended to benefit both software developers and users. Software developers could release a single binary for all platforms, and software users would have freedom to procure multiple vendors' hardware competitively.^[1] Programming language designers and implementors were also interested because standard installers would mean that only a single language front end would need to be developed.^[3]

OSF released several development 'snapshots' of ANDF, but it was never released commercially by OSF or any of its members. Various reasons have been proposed for this: for example, that having multiple installation systems would complicate software support.^[4]

After OSF stopped working on ANDF, development continued at other organizations.

Notes

[1] Rolf Jester, "Life, the Universe and Open Systems", *Australian UNIX systems User Group Newsletter* 13:1 (February 1992), p. 63f

[2] Martin Marshall, "OSF Pushes for Single Distribution Standard", *InfoWorld*, May 8, 1989, p. 5

[3] B.A. Wichmann and J. McHugh, "Ada 9X Safety and Security Annex" in W.J. Taylor, ed., *Ada in Transition*, 1992, p. 52f

[4] René J. Chevance, *Server architectures: multiprocessors, clusters, parallel systems, Web servers, and storage solutions*, Elsevier Digital Press, 2005, p. 66

Bibliography

- Stavros Macrakis, "The Structure of ANDF: Principles and Examples", Open Software Foundation, RI-ANDF-RP1-1, January, 1992.
- Stavros Macrakis, "Protecting Source Code with ANDF", Open Software Foundation, November, 1992.
- Open Systems Foundation. "OSF Architecture-Neutral Distribution Format Rationale", June 1991.

- Open Systems Foundation. "A Brief Introduction to ANDF", January 1993. Available at Google Groups (http://groups.google.com/group/comp.lang.c++/browse_frm/thread/7ce81dfc11fe770b/ec34fa84c4b61ff7?lnk=st&q=A+Brief+Introduction+to+ANDF+&rnum=3#ec34fa84c4b61ff7)

Development techniques

Bootstrapping

In computer science, **bootstrapping** is the process of writing a compiler (or assembler) in the target programming language which it is intended to compile. Applying this technique leads to a self-hosting compiler.

A large proportion of programming languages are bootstrapped, including BASIC, C, Pascal, Factor, Haskell, Modula-2, Oberon, OCaml, Common Lisp, Scheme, Python and more.

Advantages

Bootstrapping a compiler has the following advantages:^[1] ^[2]

- it is a non-trivial test of the language being compiled;
- compiler developers only need to know the language being compiled;
- compiler development can be done in the higher level language being compiled;
- improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself; and
- it is a comprehensive consistency check as it should be able to reproduce its own object code.

The chicken and egg problem

If one needs a compiler for language X to obtain a compiler for language X (which is written in language X), how did the first compiler get written? Possible methods to solving this chicken and egg problem include:

- Implementing an interpreter or compiler for language X in language Y. Niklaus Wirth reported that he wrote the first Pascal compiler in Fortran.
- Another interpreter or compiler for X has already been written in another language Y; this is how Scheme is often bootstrapped.
- Earlier versions of the compiler were written in a subset of X for which there existed some other compiler; this is how some supersets of Java, Haskell, and the initial Free Pascal compiler are bootstrapped.
- The compiler for X is cross compiled from another architecture where there exists a compiler for X; this is how compilers for C are usually ported to other platforms. Also this is the method used for Free Pascal after the initial bootstrap.
- Writing the compiler in X; then hand-compiling it from source (most likely in a non-optimized way) and running that on the code to get an optimized compiler. Donald Knuth used this for his WEB literate programming system.

Methods for distributing compilers in source code include providing a portable bytecode version of the compiler, so as to *bootstrap* the process of compiling the compiler with itself.

Such methods are also one way of detecting or avoiding (or both) the potential problem pointed out in *Reflections on Trusting Trust*.

The T-diagram is a notation used to explain these compiler bootstrap techniques.^[2]

In some cases, the most convenient way to get a complicated compiler running on a system that has little or no software on it involves a series of ever more sophisticated assemblers and compilers.^[3]

History

The first language to provide such a bootstrap was NELIAC. The first commercial language to do so was PL/I.

The first self-hosting compiler (excluding assemblers) was written for Lisp by Hart and Levin at MIT in 1962. They wrote a Lisp compiler in Lisp, testing it inside an existing Lisp interpreter. Once they had improved the compiler to the point where it could compile its own source code, it was self-hosting.^[4]

The compiler as it exists on the standard compiler tape is a machine language program that was obtained by having the S-expression definition of the compiler work on itself through the interpreter. (AI Memo 39)^[4]

This technique is only possible when an interpreter already exists for the very same language that is to be compiled. It borrows directly from the notion of running a program on itself as input, which is also used in various proofs in theoretical computer science, such as the proof that the halting problem is undecidable.

References

- [1] Compilers and Compiler Generators: An Introduction With C++. Patrick D. Terry 1997. International Thomson Computer Press. ISBN 1850322988
- [2] "Compiler Construction and Bootstrapping" by P.D.Terry 2000. HTML (<http://www.oopweb.com/Compilers/Documents/Compilers/Volume/cha03s.htm>). PDF (<http://webster.cs.ucr.edu/AsmTools/RollYourOwn/CompilerBook/CHAP03.PDF>). HTML (<http://cm.bell-labs.com/who/ken/trust.html>).
- [3] "Bootstrapping a simple compiler from nothing" (<http://homepage.ntlworld.com/edmund.grimley-evans/bcompiler.html>) by Edmund GRIMLEY EVANS 2001
- [4] Tim Hart and Mike Levin. "AI Memo 39-The new compiler" (<ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf>). . Retrieved 2008-05-23.

Compiler correctness

In computing, **compiler correctness** is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification. Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Formal methods

Compiler validation with formal methods involves a long chain of formal, deductive logic.^[1] However, since the tool to find the proof (theorem prover) is implemented in software and is complex, there is a high probability it will contain errors. One approach has been to use a tool that verifies the proof (a proof checker) which because it is much simpler than a proof-finder is less likely to contain errors.

A language described as a subset of C has been formally verified (although no proof was given of its connection to the C Standard), and the proof has been machine checked.^[2]

Methods include model checking and formal verification.

Testing

Testing represents a significant portion of the effort in shipping a compiler, but receives comparatively little coverage in the standard literature. The 1986 edition of Aho, Sethi, & Ullman has a single-page section on compiler testing, with no named examples.^[3] The 2006 edition omits the section on testing, but does emphasize its importance: "Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."^[4] Fraser & Hanson 1995 has a brief section on regression testing; source code is available.^[5] Bailey & Davidson 2003 cover testing of procedure calls^[6] A number of articles confirm that many released compilers have significant

code-correctness bugs.^[7] Sheridan 2007 is probably the most recent journal article on general compiler testing.^[8] Commercial compiler compliance validation suites are available from ACE^[9], Perennial^[10], and Plum-Hall^[11]. For most purposes, the largest body of information available on compiler testing are the Fortran^[12] and Cobol^[13] validation suites.

References

- [1] doi:10.1145/359104.359106
- [2] doi:10.1145/1111320.1111042
- [3] *Compilers: Principles, Techniques and Tools, infra* 1986, p. 731.
- [4] *ibid*, 2006, p. 16.
- [5] Christopher Fraser; David Hanson (1995). *A Retargetable C compiler: Design and Implementation*. Benjamin/Cummings Publishing. ISBN 0-8053-1670-1., pp. 531–3.
- [6] Mark W. Bailey; Jack W. Davidson (2003). "Automatic Detection and Diagnosis of Faults in Generated Code for Procedure Calls" (<http://big-oh.cs.hamilton.edu/~bailey/pubs/techreps/TR-2001-1.pdf>). *IEEE Transactions on Software Engineering* **29** (11). . Retrieved 2009-03-24., p. 1040.
- [7] E.g., Christian Lindig (2005). "Random testing of C calling conventions" (<http://quest-tester.googlecode.com/svn/trunk/doc/lindig-aadebug-2005.pdf>). *Proceedings of the Sixth International Workshop on Automated Debugging*. ACM. ISBN 1-59593-050-7.. Retrieved 2009-03-24., and Eric Eide; John Regehr (2008). "Volatile are miscompiled, and what to do about it" (<http://www.cs.utah.edu/~regehr/papers/emsoft08-preprint.pdf>). *Proceedings of the 7th ACM international conference on Embedded software*. ACM. ISBN 978-1-60558-468-3.. Retrieved 2009-03-24.
- [8] Flash Sheridan (2007). "Practical Testing of a C99 Compiler Using Output Comparison" (http://pobox.com/~flash/Practical_Testing_of_C99.pdf). *Software: Practice and Experience* **37** (14): 1475–1488. doi:10.1002/spe.812. . Retrieved 2009-03-24. Bibliography at "" (http://pobox.com/~flash/compiler_testing_bibliography.html). . Retrieved 2009-03-13..
- [9] "" (<http://www.ace.nl/compiler/supertest.html>). . Retrieved 2011-01-15.
- [10] "" (http://peren.com/pages/products_set.htm). . Retrieved 2009-03-13.
- [11] "" (<http://plumhall.com/suites.html>). . Retrieved 2009-03-13.
- [12] "Source of Fortran validation suite" (http://www.itl.nist.gov/div897/ctg/fortran_form.htm). . Retrieved 2011-09-01.
- [13] "Source of Cobol validation suite" (http://www.itl.nist.gov/div897/ctg/cobol_form.htm). . Retrieved 2011-09-01.

Jensen's Device

Jensen's Device is a computer programming technique devised by Danish computer scientist Jørn Jensen, who worked with Peter Naur at Regnecentralen, particularly on the GIER Algol compiler, one of the earliest correct implementations of ALGOL 60.^[1]

The following program was proposed to illustrate the technique. It computes the 100th harmonic number by the formula $H_{100} = \sum_{i=1}^{100} \frac{1}{i}$:

```

begin
  integer i;
  real procedure sum (i, lo, hi, term);
    value lo, hi;
    integer i, lo, hi;
    real term;
    comment term is passed by-name, and so is i;
  begin
    real temp;
    temp := 0;
    for i := lo step 1 until hi do
      temp := temp + term;
      sum := temp
    end;
    comment note the correspondence between the mathematical notation and the call to sum;
    print (sum (i, 1, 100, 1/i))
  end

```

The above exploits call by name to produce the correct answer (5.187...). It depends on the assumption that an expression passed as an actual parameter to a procedure would be re-evaluated every time the corresponding formal parameter's value was required. Thus, assignment of a value to *i* in the *sum* routine as a part of the *for* statement changes the value of the original *i* variable, and when the code of the *for* loop requires the value of *term*, the expression $1/i$ is evaluated and with the new value of *i*. If the last parameter to *sum* had been passed by *value*, and assuming the initial value of *i* were 1, the result would have been $100 \times 1/1 = 100$, though actually, it is not initialised.

So, the *first* parameter to *sum*, representing the "bound" variable of the summation, must also be passed by name, otherwise it would not be possible to compute the values to be added. On the other hand, the global variable does not have to use the same identifier, in this case *i*, as the formal parameter.

Donald Knuth later proposed the Man or Boy Test as a more rigorous exercise.

Usage

The *Sum* function can be employed for arbitrary functions merely by employing the appropriate expressions. If a sum of integers were desired the expression would be just *sum(i, 1, 100, i)*; if a sum of squares of integers, then *sum(i, 1, 100, i*i)*; and so on. A slight variation would be suitable for initiating a numerical integration of an expression by a method very similar to that of *sum*. In the absence of this pass-by-name facility, it would be necessary to define functions embodying those expressions to be passed according to the protocols of the computer

language, or to create a compendium function along with some arrangement to select the desired expression for each usage.

Because every mention of `term` within the procedure re-evaluates the original expression, making a local copy of the value of `term` as appropriate may be worthwhile, as when the procedure might compute the product of the terms as well as their sum.

References

- [1] Peter Naur's 2005 Turing Award citation (<http://awards.acm.org/citation.cfm?id=1024454&srt=all&aw=140&ao=AMTURING>)
mentions his work with Jensen on GIER Algol

Man or boy test

The **man or boy test** was proposed by computer scientist Donald Knuth as a means of evaluating implementations of the ALGOL 60 programming language. The aim of the test was to distinguish compilers that correctly implemented "recursion and non-local references" from those that did not.

There are quite a few ALGOL60 translators in existence which have been designed to handle recursion and non-local references properly, and I thought perhaps a little test-program may be of value. Hence I have written the following simple routine, which may separate the "man-compilers" from the "boy-compilers".

— Donald Knuth^[1]

Knuth's example

```
begin
  real procedure A(k, x1, x2, x3, x4, x5);
  value k; integer k;
begin
  real procedure B;
  begin k := k - 1;
    B := A := A(k, B, x1, x2, x3, x4);
  end;
  if k <= 0 then A := x4 + x5 else B;
  end;
  outreal(A(10, 1, -1, -1, 1, 0));
end;
```

This creates a tree of *B* call frames that refer to each other and to the containing *A* call frames, each of which has its own copy of *k* that changes every time the associated *B* is called. Trying to work it through on paper is probably fruitless, but the correct answer is -67, despite the fact that in the original paper Knuth conjectured it to be -121. The survey paper by Charles H. Lindsey mentioned in the references contains a table for different starting values. Even modern machines quickly run out of stack space for larger values of *k*.^[2]

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	1	0	-2	0	1	0	1	-1	-10	-30	-67	-138	-291	-642	-1446	-3250	-7244	-16065	-35601	-78985	-175416	-389695	-865609	-1922362	-4268854	-9479595

Explanation

There are three Algol features used in this program that can be difficult to implement properly in a compiler:

1. **Nested function definitions:** Since B is being defined in the local context of A , the body of B has access to symbols that are local to A — most notably k which it modifies, but also $x1, x2, x3, x4$, and $x5$. This is straightforward in the Algol descendant Pascal, but not possible in the other major Algol descendant C (although C's address-of operator & makes it quite possible to pass around pointers to local variables in arbitrary functions, so this can be worked around).
2. **Function references:** The B in the recursive call $A(k, B, x1, x2, x3, x4)$ is not a call to B , but a reference to B , which will be called only when it appears as $x4$ or $x5$ in the statement $A := x4 + x5$. This is conversely straightforward in C, but not possible in many dialects of Pascal (although the ISO 7185 standard supports function parameters). When the set of functions that may be referenced is known beforehand (in this program it is only B), this can be worked around.
3. **Constant/function dualism:** The $x1$ through $x5$ parameters of A may be numeric constants or references to the function B — the $x4 + x5$ expression must be prepared to handle both cases as if the formal parameters $x4$ and $x5$ had been replaced by the corresponding actual parameter (call by name). This is probably more of a problem in statically typed languages than in dynamically typed languages, but the standard work-around is to reinterpret the constants 1, 0, and -1 in the main call to A as functions without arguments that return these values.

These things are however not what the test is about; they're merely prerequisites for the test to at all be meaningful. What the test is *about* is whether the different references to B resolve to the *correct* instance of B — one which has access to the same A -local symbols as the B which created the reference. A "boy" compiler might for example instead compile the program so that B always accesses the topmost A call frame.

References

- [1] Donald Knuth (July 1964). "Man or boy?" (<http://www.chilton-computing.org.uk/acl/applications/algol/p006.htm>). . Retrieved Dec 25, 2009.
- [2] See Performance and Memory on the Rosetta Code Man or Boy Page (http://rosettacode.org/wiki/Man_or_boy_test)

External links

- The Man or Boy Test (<http://portal.acm.org/toc.cfm?id=1060969>) as published in the *ALGOL Bulletin 17*, p7 (available at chilton-computing.org (<http://www.chilton-computing.org.uk/acl/applications/algol/p006.htm>))
- Man or boy test (http://www.rosettacode.org/wiki/Man_or_boy_test) examples in many programming languages

Cross compiler

A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is run. Cross compiler tools are used to generate executables for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system. It has become more common to use this tool for paravirtualization where a system may have one or more platforms in use.

Not targeted by this definition are source to source translators, which are often mistakenly called cross compilers.

Uses of cross compilers

The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in a number of situations:

- Embedded computers where a device has extremely limited resources. For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food. This computer will not be powerful enough to run a compiler, a file system, or a development environment. Since debugging and testing may also require more resources than are available on an embedded system, cross-compilation can be less involved and less prone to errors than native compilation.
- Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm. Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across any machine that is free, regardless of its underlying hardware or the operating system version that it is running.
- Bootstrapping to a new platform. When developing software for a new platform, or the emulator of a future platform, one uses a cross compiler to compile necessary tools such as the operating system and a native compiler.
- Compiling native code for emulators for older now-obsolete platforms like the Commodore 64 or Apple II by enthusiasts who use cross compilers that run on a current platform (such as Aztec C's MS DOS 6502 cross compilers running under Windows XP).

Use of virtual machines (such as Java's JVM) resolves some of the reasons for which cross compilers were developed. The virtual machine paradigm allows the same compiler output to be used across multiple target systems, although this is not always ideal because virtual machines are often slower and the compiled program can only be run on computers with that virtual machine.

Typically the hardware architecture differs (e.g. compiling a program destined for the MIPS architecture on an x86 computer) but cross-compilation is also applicable when only the operating system environment differs, as when compiling a FreeBSD program under Linux, or even just the system library, as when compiling programs with uClibc on a glibc host.

Canadian Cross

The **Canadian Cross** is a technique for building cross compilers for other machines. Given three machines A, B, and C, one uses machine A (e.g. running Windows XP on an IA-32 processor) to build a cross compiler that runs on machine B (e.g. running Mac OS X on an x86-64 processor) to create executables for machine C (e.g. running Android on an ARM processor). When using the Canadian Cross with GCC, there may be four compilers involved:

- The *proprietary (describes goods which are made and sent out by a particular company whose name is on the product) native Compiler for machine A (1)* (e.g. compiler from Microsoft Visual Studio) is used to build the *gcc native compiler for machine A (2)*.
- The *gcc native compiler for machine A (2)* is used to build the *gcc cross compiler from machine A to machine B (3)*
- The *gcc cross compiler from machine A to machine B (3)* is used to build the *gcc cross compiler from machine B to machine C (4)*

The end-result cross compiler (4) will not be able to run on your build machine A; instead you would use it on machine B to compile an application into executable code that would then be copied to machine C and executed on machine C.

For instance, NetBSD provides a POSIX Unix shell script named `build.sh` which will first build its own toolchain with the host's compiler; this, in turn, will be used to build the cross-compiler which will be used to build the whole system.

The term **Canadian Cross** came about because at the time that these issues were all being hashed out, Canada had three national political parties.^[1]

Time line of early cross compilers

- 1979 – ALGOL 68C generated ZCODE, this added on porting the compiler and other ALGOL 68 applications to alternate platforms. To compile the ALGOL 68C compiler required about 120kB of memory. With Z80 its 64kB memory is too small to actually compile the compiler. So for the Z80 the compiler itself had to be cross compiled from the larger CAP capability computer or an IBM 370 mainframe.

GCC and cross compilation

GCC, a free software collection of compilers, can be set up to cross compile. It supports many platforms and languages.

GCC requires that a compiled copy of binutils be available for each targeted platform. Especially important is the GNU Assembler. Therefore, binutils first has to be compiled correctly with the switch `--target=some-target` sent to the configure script. GCC also has to be configured with the same `--target` option. GCC can then be run normally provided that the tools, which binutils creates, are available in the path, which can be done using the following (on UNIX-like operating systems with bash):

```
PATH=/path/to/binutils/bin:$PATH make
```

Cross compiling GCC requires that a portion of the *target platform's* C standard library be available on the *host platform*. At least the crt0, ... components of the library must be available. You may choose to compile the full C library, but that can be too large for many platforms. The alternative is to use newlib, which is a small C library containing only the most essential components required to compile C source code. To configure GCC with newlib, use the switch `--with-newlib`.

The GNU autotools packages (i.e. autoconf, automake, and libtool) use the notion of a *build platform*, a *host platform*, and a *target platform*. The *build platform* is where the compiler is actually compiled. In most cases, build should be left undefined (it will default from host). The *host platform* is where the output artefacts from the compiler

will be executed. The *target platform* is used when cross compiling cross compilers, it represents what type of object code the package itself will produce; otherwise the *target platform* setting is irrelevant.^[2] For example, consider cross-compiling a video game that will run on a Dreamcast. The machine where the game is compiled is the *host platform* while the Dreamcast is the *target platform*.

Another method that is popularly used by embedded Linux developers is to use gcc, g++, gjc etc. with scratchbox or the newer scratchbox2. These tools create a "chroot"ed sandbox where you can build up your tools, libc, and libraries without having to set extra paths. It also has facilities for tricking the runtime into thinking it is on (for example) an ARM CPU so things like configure scripts will run. The downside to scratchbox is that it is slower and you lose access to most of your tools that are on the host. The speed loss is not terrible and you can move host tools into scratchbox.

Manx Aztec C cross compilers

Manx Software Systems, of Shrewsbury, New Jersey, produced C compilers beginning in the 1980s targeted at professional developers for a variety of platforms up to and including PCs and Macs.

Manx's Aztec C programming language was available for a variety of platforms including MS DOS, Apple II DOS 3.3 and ProDOS, Commodore 64, Macintosh 68XXX^[3] and Amiga.

From the 1980s and continuing throughout the 1990s until Manx Software Systems disappeared, the MS DOS version of Aztec C^[4] was offered both as a native mode compiler or as a cross compiler for other platforms with different processors including the Commodore 64^[5] and Apple II.^[6] Internet distributions still exist for Aztec C including their MS DOS based cross compilers. They are still in use today.

Manx's Aztec C86, their native mode 8086 MS DOS compiler, was also a cross compiler. Although it did not compile code for a different processor like their Aztec C65 6502 cross compilers for the Commodore 64 and Apple II, it created binary executables for then-legacy operating systems for the 16 bit 8086 family of processors.

When the IBM PC was first introduced it was available with a choice of operating systems, CP/M 86 and PC DOS being two of them. Aztec C86 was provided with link libraries for generating code for both IBM PC operating systems. Throughout the 1980s later versions of Aztec C86 (3.xx, 4.xx and 5.xx) added support for MS DOS "transitory" versions 1 and 2^[7] and which were less robust than the "baseline" MS DOS version 3 and later which Aztec C86 targeted until its demise.

Finally, Aztec C86 provided C language developers with the ability to produce ROM-able "HEX" code which could then be transferred using a ROM Burner directly to an 8086 based processor. Paravirtualization may be more common today but the practice of creating low-level ROM code was more common per-capita during those years when device driver development was often done by application programmers for individual applications, and new devices amounted to a cottage industry. It was not uncommon for application programmers to interface directly with hardware without support from the manufacturer. This practice was similar to Embedded Systems Development today.

Thomas Fenwick and James Goodnow II were the two principal developers of Aztec-C. Fenwick later became notable as the author of the Microsoft Windows CE Kernel or NK ("New Kernel") as it was then called.^[8]

Microsoft C cross compilers

Early History – 1980s

Microsoft C (MSC) has a long history^[9] dating back to the 1980s. The first Microsoft C Compilers were made by the same company who made Lattice C and were rebranded by Microsoft as their own, until MSC 4 was released, which was the first version that Microsoft produced themselves.^[10]

In 1987 many developers started switching to Microsoft C, and many more would follow throughout the development of Microsoft Windows to its present state. Products like Clipper and later Clarion emerged that offered easy database application development by using cross language techniques, allowing part of their programs to be compiled with Microsoft C.

1987

C programs had long been linked with modules written in assembly language. Most C compilers (even current compilers) offer an assembly language pass (that can be tweaked for efficiency then linked to the rest of the program after assembling).

Compilers like Aztec-C converted everything to assembly language as a distinct pass and then assembled the code in a distinct pass, and were noted for their very efficient and small code, but by 1987 the optimizer built into Microsoft C was very good, and only "mission critical" parts of a program were usually considered for rewriting. In fact, C language programming had taken over as the "lowest-level" language, with programming becoming a multi-disciplinary growth industry and projects becoming larger, with programmers writing user interfaces and database interfaces in higher-level languages, and a need had emerged for cross language development that continues to this day.

By 1987, with the release of MSC 5.1, Microsoft offered a cross language development environment for MS DOS. 16 bit binary object code written in assembly language (MASM) and Microsoft's other languages including Quick Basic, Pascal, and Fortran could be linked together into one program, in a process they called "Mixed Language Programming" and now "InterLanguage Calling".^[11] If BASIC was used in this mix, the main program needed to be in BASIC to support the internal run-time system that compiled BASIC required for garbage collection and its other managed operations that simulated a BASIC interpreter like QBasic in MS DOS.

The calling convention for C code in particular was to pass parameters in "reverse order" on the stack and return values on the stack rather than in a processor register. There were other programming rules to make all the languages work together, but this particular rule persisted through the cross language development that continued throughout Windows 16 and 32 bit versions and in the development of programs for OS 2, and which persists to this day. It is known as the Pascal calling convention.

Another type of cross compilation that Microsoft C was used for during this time was in retail applications that require handheld devices like the Symbol Technologies PDT3100 (used to take inventory), which provided a link library targeted at an 8088 based barcode reader. The application was built on the host computer then transferred to the handheld device (via a serial cable) where it was run, similar to what is done today for that same market using Windows Mobile by companies like Motorola, who bought Symbol.

Early 1990s

Throughout the 1990s and beginning with MSC 6 (their first ANSI C compliant compiler) Microsoft re-focused their C compilers on the emerging Windows market, and also on OS 2 and in the development of GUI programs. Mixed language compatibility remained through MSC 6 on the MS DOS side, but the API for Microsoft Windows 3.0 and 3.1 was written in MSC 6. MSC 6 was also extended to provide support for 32-bit assemblies and support for the emerging Windows for Workgroups and Windows NT which would form the foundation for Windows XP. A

programming practice called a thunk was introduced to allow passing between 16 and 32 bit programs that took advantage of runtime binding (dynamic linking) rather than the static binding that was favoured in monolithic 16 bit MS DOS applications. Static binding is still favoured by some native code developers but does not generally provide the degree of code reuse required by newer best practices like the Capability Maturity Model (CMM).

MS DOS support was still provided with the release of Microsoft's first C++ Compiler, MSC 7, which was backwardly compatible with the C programming language and MS DOS and supported both 16 bit and 32 bit code generation.

MSC took over where Aztec C86 left off. The market share for C compilers had turned to cross compilers which took advantage of the latest and greatest Windows features, offered C and C++ in a single bundle, and still supported MS DOS systems that were already a decade old, and the smaller companies that produced compilers like Aztec C could no longer compete and either turned to niche markets like embedded systems or disappeared.

MS DOS and 16 bit code generation support continued until MSC 8.00c which was bundled with Microsoft C++ and Microsoft Application Studio 1.5, the forerunner of Microsoft Visual Studio which is the cross development environment that Microsoft provide today.

Late 1990s

MSC 12 was released with Microsoft Visual Studio 6 and no longer provided support for MS DOS 16 bit binaries, instead providing support for 32 bit console applications, but provided support for Windows 95 and Windows 98 code generation as well as for Windows NT. Link libraries were available for other processors that ran Microsoft Windows; a practice that Microsoft continues to this day.

MSC 13 was released with Visual Studio 2003, and MSC 14 was released with Visual Studio 2005, both of which still produce code for older systems like Windows 95, but which will produce code for several target platforms including the mobile market and the ARM architecture.

.NET and beyond

In 2001 Microsoft developed the Common Language Runtime (CLR), which formed the core for their .NET Framework compiler in the Visual Studio IDE. This layer on the operating system which is in the API allows the mixing of development languages compiled across platforms that run the Windows operating system.

The .NET Framework runtime and CLR provide a mapping layer to the core routines for the processor and the devices on the target computer. The command-line C compiler in Visual Studio will compile native code for a variety of processors and can be used to build the core routines themselves.

Microsoft .NET applications for target platforms like Windows Mobile on the ARM architecture cross-compile on Windows machines with a variety of processors and Microsoft also offer emulators and remote deployment environments that require very little configuration, unlike the cross compilers in days gone by or on other platforms.

Runtime libraries, such as Mono, provide compatibility for cross-compiled .NET programs to other operating systems, such as Linux.

Libraries like Qt and its predecessors including XVT provide source code level cross development capability with other platforms, while still using Microsoft C to build the Windows versions. Other compilers like MinGW have also become popular in this area since they are more directly compatible with the Unixes that comprise the non-Windows side of software development allowing those developers to target all platforms using a familiar build environment.

Free Pascal

Free Pascal was developed from the beginning as a cross compiler. The compiler executable (ppcXXX where XXX is a target architecture) is capable of producing executables (or just object files if no internal linker exists, or even just assembly files if no internal assembler exists) for all OS of the same architecture. For example, ppc386 is capable of producing executables for i386-linux, i386-win32, i386-go32v2 (DOS) and all other OSes (see ^[12]). For compiling to another architecture, however, a cross architecture version of the compiler must be built first. The resulting compiler executable would have additional 'ross' before the target architecture in its name. i.e if the compiler is built to target x64, then the executable would be ppcrossx64.

To compile for a chosen architecture-OS, the compiler switch (for the compiler driver fpc) -P and -T can be used. This is also done when cross compiling the compiler itself, but is set via make option CPU_TARGET and OS_TARGET. GNU assembler and linker for the target platform is required if Free Pascal doesn't yet have internal version of the tools for the target platform.

References

- [1] "4.9 Canadian Crosses" (<http://vmlinux.org/crash/mirror/www.objsw.com/CrossGCC/FAQ-4.html>). *CrossGCC*. . Retrieved 2007-10-11. "This is called a 'Canadian Cross' because at the time a name was needed, Canada had three national parties."
- [2] http://www.gnu.org/s/libtool/manual/automake/Cross_002dCompilation.html
- [3] Obsolete Macintosh Computers (<http://docs.info.apple.com/article.html?artnum=304210>)
- [4] Aztec C (<http://www.clipshop.ca/Aztec/index.htm>)
- [5] Commodore 64 (<http://www.clipshop.ca/Aztec/index.htm#commodore>)
- [6] Apple II (<http://www.clipshop.ca/Aztec/index.htm#apple>)
- [7] MS DOS Timeline (<http://members.fortunecity.com/pcmuseum/dos.htm>)
- [8] Inside Windows CE (search for Fenwick) (<http://www.amazon.com/Inside-Microsoft-Windows-John-Murray/dp/1199000361>)
- [9] Microsoft Language Utility Version History (<http://support.microsoft.com/kb/93400>)
- [10] History of PC based C-compilers (<http://www.itee.uq.edu.au/~csmweb/decompilation/hist-c-pc.html>)
- [11] Which Basic Versions Can CALL C, FORTRAN, Pascal, MASM (<http://support.microsoft.com/kb/35965>)
- [12] "Free Pascal Supported Platform List" (http://wiki.lazarus.freepascal.org/Platform_list). *Platform List*. . Retrieved 2010-06-17. "i386"

External links

- Cross Compilation Tools (http://www.airs.com/ian/configure/configure_5.html) – reference for configuring GNU cross compilation tools
- Building Cross Toolchains with gcc (http://gcc.gnu.org/wiki/Building_Cross_Toolchains_with_gcc) is a wiki of other GCC cross-compilation references
- Scratchbox (<http://www.scratchbox.org/>) is a toolkit for Linux cross-compilation to ARM and x86 targets
- Grand Unified Builder (GUB) (<http://lilypond.org/gub/>) for Linux to cross-compile multiple architectures e.g.: Win32/Mac OS/FreeBSD/Linux used by GNU LilyPond
- Crosstool (<http://kegel.com/crosstool/>) is a helpful toolchain of scripts, which create a Linux cross-compile environment for the desired architecture, including embedded systems
- crosstool-NG (<http://crosstool-ng.org/>) is a rewrite of Crosstool and helps building toolchains.
- buildroot (<http://buildroot.uclibc.org/>) is another set of scripts for building a uClibc-based toolchain, usually for embedded systems. It is utilized by OpenWrt.
- ELDK (Embedded Linux Development Kit) (<http://www.denx.de/wiki/ELDK-5/WebHome>). Utilized by Das U-Boot.
- T2 SDE (<http://t2-project.org/>) is another set of scripts for building whole Linux Systems based on either GNU libC, uClibc or dietlibc for a variety of architectures
- Cross Linux from Scratch Project (<http://trac.cross-lfs.org/>)
- IBM has a very clear structured tutorial (<https://www6.software.ibm.com/developerworks/education/l-cross-l-cross-ltr.pdf>) about cross-building a GCC toolchain.

- (French) Cross-compilation avec GCC 4 sous Windows pour Linux (<http://tcuvelier.developpez.com/tutoriels/cross-gcc/gcc-cross/>) - A tutorial to build a cross-GCC toolchain, but from Windows to Linux, a subject rarely developed

Source-to-source compiler

A **source-to-source compiler** is a type of compiler that takes a high level programming language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g., OpenMP) or language constructs (e.g. Fortran's DOALL statements)^[1].

Another purpose of source-to-source-compiling is translating legacy code to use the next version of the underlying programming language or an API that breaks backward compatibility. It will perform automatic code refactoring which is useful when the programs to refactor are outside the control of the original implementer (for example, converting programs from Python 2 to Python 3, or converting programs from an old API to the new API) or when the size of the program makes it impractical or time consuming to refactor it by hand.

Examples

DMS Software Reengineering Toolkit

DMS Software Reengineering Toolkit is a source-to-source program transformation tool, parameterized by explicit source and target (may be the same) computer language definitions. It can be used for translating from one computer language to another, for compiling domain-specific languages to a general purpose language, or for carrying out optimizations or massive modifications within a specific language. DMS has a library of language definitions for most widely used computer languages (including full C++, and a means for defining other languages which it does not presently know).

LLVM

Low Level Virtual Machine (LLVM) can translate from any language supported by gcc 4.2.1 (Ada, C, C++, Fortran, Java, Objective-C, or Objective-C++) or by clang to any of: C, C++, or MSIL by way of the "arch" command in llvm-gcc.

```
% llvm-g++ x.cpp -o program
% llc -march=c program.bc -o program.c
% cc x.c

% llvm-g++ x.cpp -o program
% llc -march=msil program.bc -o program.msil
```

Refactoring tools

The refactoring tools automate transforming source code into another:

- The Python 3000 2to3 tool transform non forward-compatible Python 2 code into Python 3 code
- Qt's qt3to4^[2] tool convert non forward-compatible usage of the Qt3 API into Qt4 API usage.
- Coccinelle uses semantic patches to describe refactoring to apply to C code. It's been applied successfully to refactor the drivers of the Linux kernel due to kernel API changes^[3].
- RefactoringNG^[4] is a Netbeans module for refactoring Java code where you can write transformations rules of a program's abstract syntax tree.

Vala

Vala, as another example, is compiled to C (with additional libraries such as GObject) before ever being compiled to native code.

References

- [1] "Types of compilers" (<http://www.compilers.net/paedia/compiler/index.htm>). compilers.net. 1997-2005. . Retrieved 28 October 2010.
- [2] <http://doc.trolltech.com/4.0/qt3to4.html#qt3to4>
- [3] Valerie Henson (January 20, 2009). "Semantic patching with Coccinelle" (<http://lwn.net/Articles/315686/>). lwn.net. . Retrieved 28 October 2010.
- [4] <http://kenai.com/projects/refactoringng/>

Tools

Compiler-compiler

A **compiler-compiler** or **compiler generator** is a tool that creates a parser, interpreter, or compiler from some form of formal description of a language and machine. The earliest and still most common form of compiler-compiler is a **parser generator**, whose input is a grammar (usually in BNF) of a programming language, and whose generated output is the source code of a parser often used as a component of a compiler. Similarly, **code generator-generators** such as (JBurg^[1]) exist, but such tools have not yet reached maturity.

The ideal compiler-compiler takes a description of a programming language and a target instruction set architecture, and automatically generates a usable compiler from them. In practice, the state of the art has yet to reach this degree of sophistication and most compiler generators are not capable of handling semantic or target architecture information.

Variants

A typical parser generator associates executable code with each of the rules of the grammar that should be executed when these rules are applied by the parser. These pieces of code are sometimes referred to as semantic action routines since they define the semantics of the syntactic structure that is analyzed by the parser. Depending upon the type of parser that should be generated, these routines may construct a parse tree (or abstract syntax tree), or generate executable code directly.

One of the earliest (1964), surprisingly powerful, versions of compiler-compilers is MetaII, which accepted grammars and code generation rules, and is able to compile itself and other languages.

Some experimental compiler-compilers take as input a formal description of programming language semantics, typically using denotational semantics. This approach is often called 'semantics-based compiling', and was pioneered by Peter Mosses' Semantic Implementation System (SIS) in 1978.^[2] However, both the generated compiler and the code it produced were inefficient in time and space. No production compilers are currently built in this way, but research continues.

The Production Quality Compiler-Compiler project at Carnegie-Mellon University does not formalize semantics, but does have a semi-formal framework for machine description.

Compiler-compilers exist in many flavors, including bottom-up rewrite machine generators (see JBurg^[1]) used to tile syntax trees according to a rewrite grammar for code generation, and attribute grammar parser generators (e.g. ANTLR can be used for simultaneous type checking, constant propagation, and more during the parsing stage).

History

The first compiler-compiler to use that name was written by Tony Brooker in 1960 and was used to create compilers for the Atlas computer at the University of Manchester, including the Atlas Autocode compiler. However it was rather different from modern compiler-compilers, and today would probably be described as being somewhere between a highly customisable generic compiler and an extensible-syntax language. The name 'compiler-compiler' was far more appropriate for Brooker's system than it is for most modern compiler-compilers, which are more accurately described as parser generators. It is almost certain that the "Compiler Compiler" name has entered common use due to Yacc rather than Brooker's work being remembered.

Other examples of parser generators in the yacc vein are ANTLR, Coco/R, CUP, GNU bison, Eli, FSL, SableCC and JavaCC.

Several compiler-compilers

- ANTLR
- Bison
- Coco/R
- ELI, an integrated toolset for compiler construction.
- Lemon
- parboiled, a Java library for building parsers.
- Packrat parser
- PQCC, a compiler-compiler that is more than a parser generator.
- Yacc

Other related topics

- Parsing expression grammar
- LL parser
- LR parser
- Simple LR parser
- LALR parser
- GLR parser

Notes

[1] <http://jburg.sourceforge.net/>

[2] Peter Mosses, "SIS: A Compiler-Generator System Using Denotational Semantics," Report 78-4-3, Dept. of Computer Science, University of Aarhus, Denmark, June 1978

References

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

Further reading

Historical

- Brooker, R. A., et al., *The Compiler-Compiler*, Annual Review in Automatic Programming, Vol. 3, p. 229. (1963).
- Brooker, R. A., Morris, D. and Rohl, J. S., *Experience with the Compiler Compiler* (<http://comjnl.oxfordjournals.org/cgi/content/abstract/9/4/345>), Computer Journal, Vol. 9, p. 350. (February 1967).
- Johnson, Stephen C., *Yacc—yet another compiler-compiler*, Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, July 1975
- McKeeman, William Marshall; Horning, James J.; Wortman, David B., CS.toronto.edu (<http://www.cs.toronto.edu/XPL/>), *A Compiler Generator*, Englewood Cliffs, N.J.: Prentice-Hall, 1970. ISBN 0-13-155077-2

External links

- Computer50.org (http://www.computer50.org/mark1/getthomas/manchester_autocodes.html), Brooker Autocodes
- Catalog.compilertools.net (<http://catalog.compilertools.net/>), The Catalog of Compiler Construction Tools
- Labraj.uni-mb.si (<http://labraj.uni-mb.si/lisa>), Lisa
- Skenz.it (<http://www.skenz.it/traduttori/>), Jflex and Cup resources (Italian)
- Gentle.compilertools.net (<http://gentle.compilertools.net/index.html>), The Gentle Compiler Construction System
- Accent.compilertools.net (<http://accent.compilertools.net/>), Accent: a Compiler for the Entire Class of Context-Free Languages
- Grammatica.percederberg.net (<http://grammatica.percederberg.net>), an open-source parser-generator for .NET and Java

PQCC

The **Production Quality Compiler-Compiler Project** (or PQCC) was a long-term project led by William Wulf at Carnegie Mellon University to produce an industrial-strength compiler-compiler. PQCC would produce full, optimizing programming language compilers from descriptions of the programming language and the target machine. Though the goal of a fully automatic process was not realized, PQCC technology and ideas were the basis of production compilers from Intermetrics, Tartan Laboratories, and others.

Objective

The focus of the project was on the semantics and machine-dependent phases of compilation, since lexical and syntactic analysis were already well-understood. Each phase was formalized in a manner that permits expression in table-driven form. The automatic construction of the compiler then consists of deriving these tables from the semantic definitions of the language and target machine. Though this approach was largely successful for target machine description, it was less so for semantics.

Bibliography

- Benjamin M. Bros gol, "TCOLAda and the "Middle End" of the PQCC Ada compiler", *Proceedings of the ACM-SIGPLAN symposium on The ADA programming language* (1980). ISBN 0-89791-030-3. Documents part of an industrial compiler using PQCC technology.
- B.W. Leverett, R.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, W.A. Wulf, "An Overview of the Production-Quality Compiler-Compiler Project", *IEEE Computer* **13**:8:38-49 (August 1980).
- William Wulf, *The Design of an Optimizing Compiler*, Elsevier Science Ltd, 1980. ISBN 0-444-00158-1. Describes Wulf's BLISS-11 compiler, which included some PQCC ideas.

Compiler Description Language

Compiler Description Language, or CDL, is a Computer language based on affix grammars. It is very similar to Backus–Naur form(BNF) notation. It was designed for the development of compilers. It is very limited in its capabilities and control flow; and intentionally so. The benefits of these limitations are twofold. On the one hand they make possible the sophisticated data and control flow analysis used by the CDL2 optimizers resulting in extremely efficient code. The other benefit is that they foster a highly verbose naming convention. This in turn leads to programs that are to a great extent self-documenting.

The language looks a bit like Prolog (this is not surprising since both languages arose at about the same time out of work on Affix grammars). As opposed to Prolog however, control flow in CDL is deterministically based on success/failure i.e., no other alternatives are tried when the current one fails. This idea is also used in Parsing Expression Grammars.

CDL3 is the third version of the CDL language, significantly different from the previous two versions.

Short Description

The original version, designed by Cornelis H. A. Koster at the University of Nijmegen emerged in 1971 had a rather unusual concept: it had no core. A typical programming language source is translated to machine instructions or canned sequences of those instructions. Those represent the core, the most basic abstractions that the given language supports. Such primitives can be the additions of numbers, copying variables to each other and so on. CDL1 lacks such a core, it is the responsibility of the programmer to provide the primitive operations in a form that can then be turned into machine instructions by means of an assembler or a compiler for a traditional language. The CDL1 language itself has no concept of primitives, no concept of data types apart from the machine word (an abstract unit of storage - not necessarily a real machine word as such). The evaluation rules are rather similar to the Backus–Naur form syntax descriptions; in fact, writing a parser for a language described in BNF is rather simple in CDL1.

Basically, the language consists of rules. A rule can either succeed or fail. A rule consists of alternatives that are sequences of other rule invocations. A rule succeeds if any of its alternatives succeeds; these are tried in sequence. An alternative succeeds if all of its rule invocations succeed. The language provides operators to create evaluation loops without recursion (although this is not strictly necessary in CDL2 as the optimizer achieves the same effect) and some shortcuts to increase the efficiency of the otherwise recursive evaluation but the basic concept is as above. Apart from the obvious application in context-free grammar parsing, CDL is also well suited to control applications, since a lot of control applications are essentially deeply nested if-then rules.

Each CDL1 rule, while being evaluated, can act on data, which is of unspecified type. Ideally the data should not be changed unless the rule is successful (no side effects on failure). This causes problems as although this rule may succeed, the rule invoking it might still fail, in which case the data change should not take effect. It is fairly easy (albeit memory intensive) to assure the above behavior if all the data is dynamically allocated on a stack but it is rather hard when there's static data, which is often the case. The CDL2 compiler is able to flag the possible violations thanks to the requirement that the direction of parameters (input,output,input-output) and the type of rules (can fail: **test, predicate;** cannot fail: **function, action;** can have side effect: **predicate, action;** cannot have side effect: **test, function**) must be specified by the programmer.

As the rule evaluation is based on calling simpler and simpler rules, at the bottom there should be some primitive rules that do the actual work. That is where CDL1 is very surprising: it does not have those primitives. You have to provide those rules yourself. If you need addition in your program, you have to create a rule that has two input parameters and one output parameter and the output is set to be the sum of the two inputs by your code. The CDL compiler uses your code as strings (there are conventions how to refer to the input and output variables) and simply emits it as needed. If you describe your adding rule using assembly, then you will need an assembler to translate the

CDL compiler's output to machine code. If you describe all the primitive rules (macros in CDL terminology) in Pascal or C, then you need a Pascal or C compiler to run after the CDL compiler. This lack of core primitives can be very painful when you have to write a snippet of code even for the simplest single-machine-instruction operation but on the other hand it gives you very great flexibility in implementing esoteric abstract primitives acting on exotic abstract objects (the 'machine word' in CDL is more like 'unit of data storage', with no reference to the kind of data stored there). Additionally large projects made use of carefully crafted libraries of primitives. These were then replicated for each target architecture and OS allowing the production of highly efficient code for all.

To get a feel for how the language looks, here is a small code fragment adapted from the CDL2 manual:

```

ACTION quicksort + >from + >to -p -q:
    less+from+to, split+from+to+p+q,
    quicksort+from+q, quicksort+p+to;
    +
.

ACTION split + >i + >j + p> + q> -m:
    make+p+i, make+q+j, add+i+j+m, halve+m,
    (again: move up+j+p+m, move down+i+q+m,
     (less+p+q, swap item+p+q, incr+p, decr+q, *again;
     less+p+m, swap item+p+m, incr+p;
     less+m+q, swap item+q+m, decr+q;
     +) ) .

FUNCTION move up + >j + >p> + >m:
    less+j+p;
    smaller item+m+p;
    incr+p, *.

FUNCTION move down + >i + >q> + >m:
    less+q+j;
    smaller item+q+m;
    decr+q, *.

TEST less+a+b:=a < b .
FUNCTION make+a+b:=a = b .
FUNCTION add+a+b+sum:=sum=a+b .
FUNCTION halve+a:=a / 2 .
FUNCTION incr+a:=a ++ .
FUNCTION decr+a:=a -- .

TEST smaller item+i+j:=items[i]<items[j] .
ACTION swap
    items+i+j=t; items[i]=items[j]; items[j]=t .

```

The primitive operations are here defined in terms of Java (or C). This is not a complete program; we must define the Java array *items* elsewhere.

CDL2, that appeared in 1976, kept the principles of CDL1 but made the language suitable for large projects. It introduced modules, enforced data-change-only-on-success and extended the capabilities of the language somewhat.

The optimizers in the CDL2 compiler and especially in the CDL2 Laboratory (an IDE for CDL2) were world class and not just for their time. One feature of the CDL2 Laboratory optimizer is almost unique: it can perform optimizations across compilation units, i.e., treating the entire program as a single compilation.

CDL3 is a more recent language. It gave up the open-ended feature of the previous CDL versions and it provides primitives to basic arithmetic and storage access. The extremely puritan syntax of the earlier CDL versions (the number of keywords and symbols both run in single digits) have also been relaxed and some basic concepts are now expressed in syntax rather than explicit semantics. In addition, data types have been introduced to the language.

A book about the CDL1 / CDL2 language can be found in ^[1].

The description of CDL3 can be found in ^[2].

Programs Developed

The commercial MBP Cobol (a Cobol compiler for the PC) as well as the MProlog system (an industrial strength Prolog implementation that ran on numerous architectures (IBM mainframe, VAX, PDP-11, Intel 8086, etc) and OS-s (DOS/OS/CMS/BS2000, VMS/Unix, DOS/Windows/OS2)). The latter is in particular a testament to CDL2-s portability.

While most programs written with CDL have been compilers, there is at least one commercial GUI application that was developed and maintained in CDL. This application was a dental image acquisition application now owned by DEXIS. A dental office management system was also once developed in CDL.

References

[1] <http://www.cs.ru.nl/~kees/cdl3/hms.ps>

[2] <http://www.cs.ru.nl/~kees/vbcourse/ivbstuff/cdl3.pdf>

Comparison of regular expression engines

Libraries

List of regular expression libraries

	Official website	Programming language	Software license
Boost.Regex ^[1]	Boost C++ Libraries ^[2]	C++	Boost Software License ^[3]
Boost.Xpressive	Boost C++ Libraries ^[4]	C++	Boost Software License ^[3]
CL-PPCRE	Edi Weitz ^[5]	Common Lisp	BSD
cppre	Jeff Stuart ^[6]	C++	GPL
DEELX	RegExLab ^[7]	C++	"free for personal use and commercial use"
FREJ ^[8]	Fuzzy Regular Expressions ^[9] for Java	Java	LGPL
GLib/GRegex ^[10]	Marco Barisione ^[11]	C	LGPL
GRETA	Microsoft Research ^[12]	C++	?

ICU	International Components for Unicode [13]	C/C++/Java	ICU license [14]
Jakarta/Regexp	The Apache Jakarta Project [15]	Java	Apache License
JRegex	JRegex [16]	Java	BSD
Oniguruma	Kosako [17]	C	BSD
Pattwo	Stevesoft [18]	Java (compatible with Java 1.0)	LGPL
PCRE	Philip Hazel [19]	C/C++ [20]	BSD
Qt/QRegEx	[21]	C++	Qt GNU GPL v. 3.0 [22] / Qt GNU LGPL v. 2.1 [23] / Qt Commercial [24]
regex - Henry Spencer's regular expression libraries	ArgList [25]	C	BSD
re2	Google Code [26]	C++	BSD
TRE [8]	Ville Laurikari [27]	C	BSD
TPerlRegEx	TPerlRegEx VCL Component [28]	Object Pascal	MPLv1.1
TRegExpr	RegExp Studio [29]	Object Pascal	Freeware
RGX	RGX [30]	C++ based component library	P6R license [31]

[1] formerly called Regex++

[2] <http://www.boost.org/libs/regex/doc/html/index.html>

[3] http://www.boost.org/LICENSE_1_0.txt

[4] <http://boost-sandbox.sourceforge.net/libs/xpressive/doc/html/index.html>

[5] <http://weitz.de/cl-ppcre/>

[6] <http://jeff.bleugris.com/journal/projects/>

[7] <http://www.regexlab.com/en/deelx/>

[8] one of fuzzy regular expression engines

[9] <http://frej.sf.net>

[10] included since version 2.13.0

[11] <http://www.barisione.org/gregex/glib-Perl-compatible-regular-expressions.html>

[12] <http://research.microsoft.com/projects/greta/>

[13] <http://www.icu-project.org/userguide/regexp.html>

[14] <http://source.icu-project.org/repos/icu/icu/trunk/license.html>

[15] <http://jakarta.apache.org/regexp/>

[16] <http://jregex.sourceforge.net/>

[17] <http://www.geocities.jp/kosako3/oniguruma/>

[18] <http://www.javaregex.com/home.html>

[19] <http://www.pcre.org/>

[20] C++ bindings were developed by Google and became officially part of PCRE in 2006

[21] <http://doc.trolltech.com/4.7/qregexp.html>

[22] <http://www.qtsoftware.com/products/licensing/licensing#qt-gnu-gpl-v>

[23] <http://www.qtsoftware.com/products/licensing/licensing#qt-gnu-lGPL-v>

[24] <http://www.qtsoftware.com/products/licensing/licensing#qt-commercial-license>

[25] <http://arglist.com/regex/>

[26] <http://code.google.com/p/re2/>

[27] <http://laurikari.net/tre/>

[28] <http://www.regexbuddy.com/delphi.html>

[29] <http://regexpstudio.com/TRegExpr/TRegExpr.html>

- [30] <https://www.p6r.com/software/rgx.html>
 [31] <https://www.p6r.com/p6r-software-library-license-1-1.html>

Languages

List of languages and frameworks coming with regular expression support

Language	Official website	Software license	Remarks
.NET	MSDN (http://msdn2.microsoft.com/en-us/library/system.text.regularexpressions.aspx)	Proprietary	
D	D (http://www.digitalmars.com/d/index.html)	Boost Software License ^[1]	
Go	Golang.org (http://golang.org/pkg/regexp/)	BSD-style license (http://golang.org/LICENSE)	
Haskell	Haskell.org (http://haskell.org/haskellwiki/Regular_expressions)	BSD3	Not included in the language report; nor in GHC's Hierarchical Libraries
Java	Java (http://www.java.com)	GNU General Public License	REs are written as strings in source code (all backslashes must be doubled, hurting readability).
JavaScript/ECMAScript		?	Limited but REs are first-class citizens of the language with a specific / . . . /mod syntax.
Lua	Lua.org (http://www.lua.org)	MIT License	Uses a simplified, limited dialect. Can be bound to a more powerful library, like PCRE or an alternative parser like LPEG.
Objective-C (Cocoa on iOS only)	Apple (http://developer.apple.com/library/ios/#documentation/Foundation/Reference/NSRegularExpression_Class/Reference/Reference.html)	Proprietary	Currently only available on iOS 4+
OCaml	Caml (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Str.html)	LGPL	
Perl	Perl.com (http://www.perl.com/doc/manual/html/pod/perlre.html)	Artistic License or the GNU General Public License	Full, central part of the language.
PHP	PHP.net (http://www.php.net/manual/en/reference.pcre.pattern.syntax.php)	PHP License	Has two implementations, with PCRE being the more efficient (speed, functionalities).
Python	python.org (http://docs.python.org/lib/module-re.html)	Python Software Foundation License	
Ruby	ruby-doc.org (http://www.ruby-doc.org/docs/ProgrammingRuby/html/ref_c_regexp.html)	GNU Library General Public License	Ruby 1.8 and 1.9 use different engines; Ruby 1.9 integrates Oniguruma.
SAP ABAP	SAP.com (http://www.sap.com)	?	

Tcl 8.4	tcl.tk (http://www.tcl.tk/)	Tcl/Tk License (http://www.tcl.tk/software/tcltk/license.html) (Permissive, similar to BSD)	
ActionScript 3	?	?	

[1] http://www.digitalmars.com/d/2.0/phobos/std_regex.html

Language features

NOTE: An application using a library for regular expression support does not necessarily offer the full set of features of the library, e.g. GNU Grep which uses PCRE does not offer lookahead support, though PCRE does.

Part 1

Language feature comparison (part 1)

TRE	Yes	Yes	Yes	Yes	No	No	No	Yes	No
Vim (2010-05-24)	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No
RGX	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes

- [1] *Non-greedy* quantifiers match as few characters as possible, instead of the default as many. Note that many older, pre-POSIX engines were non-greedy and didn't have greedy quantifiers at all
- [2] *Shy groups*, also called *non-capturing* groups cannot be referred to with backreferences; non-capturing groups are used to speed up matching where the groups content needs not be accessed later.
- [3] *Backreferences* enable referring to previously matched groups in later parts of the regex and/or replacement string (where applicable). For instance, `(ab)+\1` matches "abab" but not "abaab"
- [4] http://www.boost.org/doc/libs/1_47_0/libs/regex/doc/html/boost_regex/syntax/perl_syntax.html#boost_regex.syntax.perl_syntax.recursive_expressions
- [5] http://www.boost.org/doc/libs/1_47_0/doc/html/xpressive/user_s_guide.html#boost_xpressive.user_s_guide.grammars_and_nested_matches.embedding_a_regex_by_reference
- [6] FREJ have no repetitive quantifiers, but have "optional" element which behaves similar to simple "?" quantifier

Part 2

Language feature comparison (part 2)

	Directives [1]	Conditionals	Atomic groups [2]	Named capture [3]	Comments	Embedded code	Partial matching	Fuzzy matching	Unicode property support (http://www.unicode.org/reports/tr18/)
Boost.Regex	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Some [4] [5]
Boost.Xpressive	Yes	No	Yes	Yes	Yes	No	Yes	No	No
CL-PPCRE	Yes	Yes	Yes	Yes	Yes	Yes	?	No	No
EmEditor	Yes	Yes	?	?	Yes	No	Yes	No	?
FREJ	No	No	Yes	Yes	Yes	No	No	Yes	?
GLib/GRegex	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Some [4] [5]
GNU Grep	Yes	Yes	?	Yes	Yes	No	?	No	No
Haskell	?	?	?	?	?	No	?	No	No
Java	Yes	No	Yes	Yes [6]	No	No	?	No	Some [5]
ICU Regex	Yes	No	Yes	No	Yes	No	No	No	Yes [7]
JGsoft	Yes	Yes	Yes	Yes	Yes	No	Yes	?	Some [5]
.NET	Yes	Yes	Yes	Yes	Yes	No	?	No	Some [5]
OCaml	No	No	No	No	No	No	?	No	No
OmniOutliner 3.6.2	?	?	?	?	No	No	?	No	?
PCRE	Yes	Yes	Yes	Yes [8]	Yes	Yes	Yes	No	Some [4] [5]
Perl	Yes	Yes	Yes	Yes [9]	Yes	Yes	No	No	Yes [7]
PHP	Yes	Yes	Yes	Yes	Yes	No	No	No	No

Python	Yes	Yes	No	Yes	Yes	No	Yes	No	No
Qt/QRegExp	No	No	No	No	No	No	Yes	No	No
re2	Yes	No	?	Yes	No	No	Yes	No	Some [5]
Ruby	Yes	No	Yes	Yes	Yes	Yes	No	No	Some [5]
TRE	Yes	No	No	No	Yes	No	No	Yes	?
Vim (2010-05-24)	Yes	No	Yes	No	No	No	Yes	No	No
RGX	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes

[1] Also known as *Flags modifiers* or *Option letters*. Example pattern: "(?i:test)"

[2] Also called *Independent sub-expressions*

[3] Similar to back references but with names instead of indices

[4] Requires optional Unicode support enabled.

[5] Supports only a subset of Unicode properties, not all of them.

[6] Available as of JDK7.

[7] Supports all Unicode properties, including non-binary properties.

[8] Available as of PCRE 7.0 (as of PCRE 4.0 with Python-like syntax (?P<name>...))

[9] Available as of perl 5.9.5

API features

API feature comparison

	Native UTF-16 support	Native UTF-8 support	Non-linear input support	Dot-matches-newline option	Anchor-matches-newline option
Boost.Regex	No	No	Yes	Yes	Yes
GLib/GRegex	No	Yes [1]	No	Yes	Yes
ICU Regex	Yes	No	Yes	Yes	Yes
Java	Yes	No	Yes	Yes	Yes
.NET	No [2]	No	Yes	Yes	Yes
PCRE	No	Yes [1]	No	Yes	Yes
Qt/QRegExp	Yes	No	No	No	No
TRE	No	?	Yes	Yes	Yes
RGX	No	No	?	Yes	Yes

[1] Requires optional Unicode support enabled.

[2] Implementation is incorrect - it treats UTF-16 code units as characters, so characters outside the BMP don't work properly. See, for example, this bug report (<http://connect.microsoft.com/VisualStudio/feedback/details/357780/extend-regex-to-process-unicode-characters-not-utf-16-code-units>).

External links

- Regular Expression Flavor Comparison (<http://www.regular-expressions.info/refflavors.html>) — Detailed comparison of the most popular regular expression flavors
- Regexp Syntax Summary (<http://www.greenend.org.uk/rjk/2002/06/regexp.html>)

Comparison of parser generators

This is a list of notable lexer generators and parser generators for various language classes.

Regular languages

Name	Website	Lexer algorithm	Output languages	Grammar, code	Development platform	License
DFASTAR	[1]	DFA matrix tables	C/C++	separate	Windows 32-bit	Proprietary
Dolphin	[2]	DFA	C++	separate	all	Proprietary
flex	[3]	DFA table driven	C	mixed	all	BSD
Alex	[4]	DFA	Haskell	mixed	all	BSD
JFlex	[5]	DFA	Java	mixed	Java Virtual Machine	GNU GPL
C# Flex	[6]	DFA	C#	mixed	.NET CLR	?
JLex	[7]	DFA	Java	mixed	Java Virtual Machine	BSD-like [8]
C# Lex	[9]	DFA	C#	mixed	.NET CLR	?
CookCC	[10]	DFA	Java	mixed	Java Virtual Machine	BSD
gelex	[11]	DFA	Eiffel	mixed	Eiffel	MIT
gplex	[12]	DFA	C#	mixed	.NET CLR	BSD-like
lex	?	DFA	C	mixed	POSIX	Proprietary, CDDL
Quex	[13]	DFA direct code	C, C++	mixed	all	modified GNU LGPL (semi-free)
Ragel	[14]	DFA	C, C++, D, Java, Objective-C, Ruby	mixed	all	GNU GPL
re2c	[15]	DFA direct code	C	mixed	all	MIT
Russ Cox's regular expression implementations	[16]	DFA, NFA	Bytecode, x86 assembly language, interpreted	separate	all	MIT
lexertl	[17]	DFA	C++		all	GNU LGPL

Deterministic context-free languages

Name	Website	Parsing algorithm	Input grammar notation	Output languages	Grammar, code	Lexer	Development platform	IDE	License
ANTLR	[18]	LL(*)	EBNF	ActionScript, Ada95, C, C++, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby	mixed	generated	Java Virtual Machine	Yes	BSD
APG	[19]	Recursive descent, Backtracking	ABNF	C, C++, JavaScript	separate	none	all	No	GNU GPL
AXE	[20]	Recursive descent	AXE	C++11	mixed	none	any platform with standard C++11 compiler	No	proprietary
Beaver	[21]	LALR(1)	?	Java	mixed	external	Java Virtual Machine	No	BSD
Bison++	[22]	LALR(1)	?	C++	mixed	external	POSIX	No	GNU GPL
Bisonc++	[23]	LALR(1)	?	C++	mixed	external	POSIX	No	GNU GPL
BtYacc	[24]	Backtracking Bottom-up	?	C++	mixed	external	all	No	Public domain
byacc	[25]	LALR(1)	YACC	C	mixed	external	all	No	Public domain
BYACC/J	[26]	LALR(1)	?	C, Java	mixed	external	all	No	Public domain
CL-Yacc	[27]	LALR(1)	?	Common Lisp	mixed	external	all	No	MIT
Coco/R	[28]	LL(k)	?	C, C++, C#, F#, Java, Ada, Pascal, Modula-2, Oberon, Ruby, Unicon, Visual Basic .NET	mixed	generated	Java Virtual Machine, .NET Framework	No	GNU GPL
CookCC	[10]	LALR(1)	?	Java	mixed	generated	Java Virtual Machine	No	BSD
CppCC	[29]	LL(k)	?	C++	mixed	generated	POSIX	No	GNU GPL
CSP	[30]	LR(1)	?	C++	separate	generated	POSIX	No	Apache License 2.0
CSTools	[31]	LALR(1)	?	C#	mixed	generated	.NET Framework	No	Proprietary

CUP	[32]	LALR(1)	?	Java	mixed	external	Java Virtual Machine	No	GNU GPL
Dragon	[33]	LR(1), LALR(1)	?	C++, Java	separate	generated	all	No	GNU GPL
eli	[34]	LALR(1)	?	C	mixed	generated	POSIX	No	GNU GPL, GNU LGPL
Essence	[35]	LR(???)	?	Scheme 48	mixed	external	all	No	BSD
eyapp	[36]	LALR(1)	?	Perl	mixed	external or generated	all	No	Perl
Frown	[37]	LALR(k)	?	Haskell 98	mixed	external	all	No	GNU GPL
geyacc	[38]	LALR(1)	?	Eiffel	mixed	external	all	No	MIT
GOLD	[39]	LALR(1)	BNF	x86 assembly language, ANSI C, C#, D, Java, Pascal, Object Pascal, Python, Visual Basic 6, Visual Basic .NET, Visual C++	separate	generated	Microsoft Windows	Yes	proprietary
GPPG	[40]	LALR(1)	YACC	C#	separate	external	Microsoft Windows	Yes	BSD
Grammatica	[41]	LL(k)	BNF dialect	C#, Java	separate	generated	Java Virtual Machine	No	GNU LGPL
HiLexed	[42]	LL(*)	EBNF or Java	Java	separate	internal	Java Virtual Machine	No	GNU LGPL
Hime Parser Generator	[43]	LR(1), LALR(1), LR(0), LR(*)	?	C#	separate	generated	.NET Framework	No	GNU LGPL
Hyacc	[44]	LR(1), LALR(1), LR(0)	YACC	C	mixed	external	all	No	GNU GPL
jacc	[45]	LALR(1)	?	Java	mixed	external	Java Virtual Machine	No	BSD
JavaCC	[46]	LL(k)	?	Java	mixed	generated	Java Virtual Machine	Yes	BSD
jay	[47]	LALR(1)	YACC	C#, Java	mixed	none	Java Virtual Machine	No	BSD
JFLAP	[48]	LL(1), LALR(1)	?	Java	?	?	Java Virtual Machine	Yes	?

JetPAG	[49]	LL(k)	?	C++	mixed	generated	all	No	GNU GPL
JS/CC	[4]	LALR(1)	?	JavaScript, JScript, ECMAScript	mixed	internal	all	Yes	Artistic
KDevelop-PG-Qt	[50]	LL(1), Backtracking, Shunting yard	?	C++	mixed	generated or external	all, KDE	No	GNU LGPL
Kelbt	[51]	Backtracking LALR(1)	?	C++	mixed	generated	POSIX	No	GNU GPL
kmyacc	[52]	LALR(1)	?	C, Java, Perl, JavaScript	mixed	external	all	No	GNU GPL
Lapg	[53]	LALR(1)	?	C, C++, C#, Java, JavaScript	mixed	generated	Java Virtual Machine	No	GNU GPL
Lemon	[54]	LALR(1)	?	C	mixed	external	all	No	Public domain
LEPL	[55]	Recursive descent	Python	Python (no generation, library)	separate	none	all	No	MPL/GNU LGPL
Lime	[56]	LALR(1)	?	PHP	mixed	external	all	No	GNU GPL
LISA	[57]	LR(?), LL(?), LALR(?), SLR(?)	?	Java	mixed	generated	Java Virtual Machine	Yes	Public domain
LPG	[58]	Backtracking LALR(k)	?	Java	mixed	generated	Java Virtual Machine	No	EPL
LLgen	[59]	LL(1)	?	C	mixed	external	POSIX	No	BSD
LLnextgen	[60]	LL(1)	?	C	mixed	external	POSIX	No	GNU GPL
LRSTAR	[61]	LALR(1) matrix tables	TBNF	C, C++	separate	lexer generator included	Windows 32-bit	Microsoft Visual C/C++	Proprietary
Menhir	[62]	LR(1)	?	Objective Caml	mixed	generated	all	No	QPL
Mini Parser Generator	[63]	?	?	Python	mixed	generated	all	No	GNU LGPL
ML-Yacc	[64]	LALR(1)	?	ML	mixed	external	all	No	?
Monkey	[65]	LR(1)	?	Java	separate	generated	Java Virtual Machine	No	GNU GPL

More Than Parsing	[66]	LL(1)	?	Java	separate	generated	Java Virtual Machine	No	GNU GPL
Msta	[67]	LALR(k), LR(k)	YACC, EBNF	C, C++	mixed	external or generated	POSIX, Cygwin	No	GNU GPL
ocamlyacc	[68]	LALR(1)	?	Objective Caml	mixed	external	all	No	QPL
olex	[69]	LL(1)	?	C++	mixed	generated	all	No	GNU GPL
Parsec	[70]	LL, Backtracking	?	Haskell	mixed	none	all	No	BSD
Parse::Yapp	[71]	LALR(1)	?	Perl	mixed	external	all	No	GNU GPL
Parser Objects	?	LL(k)	?	Java	mixed	?	Java Virtual Machine	No	zlib
PCCTS	[72]	LL	?	C, C++	?	?	all	No	?
PLY	[73]	LALR(1)	?	Python	mixed	generated	all	No	MIT License
PRECC	[74]	LL(k)	?	C	separate	generated	DOS, POSIX	No	GNU GPL
QLALR	[75]	LALR(1)	?	C++	mixed	external	all	No	GNU GPL
RPA	[76]	Backtracking Bottom-up	BNF	C (no generation, library)	separate	none	all	No	GNU GPL
SableCC	[77]	LALR(1)	?	C, C++, C#, Java, Objective Caml, Python	separate	generated	all	No	GNU LGPL
SLK	[78]	LL(k)	?	C, C++, C#, Java	separate	external	all	No	Proprietary
Spirit	[79]	Recursive descent	?	C++	mixed	internal	all	No	Boost
Styx	[80]	LALR(1)	?	C, C++	separate	generated	all	No	GNU LGPL
Sweet Parser	[81]	LALR(1)	?	C++	separate	generated	Microsoft Windows	No	zlib
Tap	[82]	LL(1)	?	C++	mixed	generated	all	No	GNU GPL
TextTransformer	[83]	LL(k)	?	C++	mixed	generated	Microsoft Windows	Yes	Proprietary

TinyPG	[84]	LL(1)	?	C#, Visual Basic	?	?	Microsoft Windows	Yes	CPOL 1.0 [85]
Toy Parser Generator	[86]	Recursive descent	?	Python	mixed	generated	all	No	GNU LGPL
TP Yacc	[87]	LALR(1)	?	Turbo Pascal	mixed	external	all	Yes	GNU GPL
Whale	[88]	LR(?), some conjunctive stuff, see Whale Calf	?	C++	mixed	external	all	No	Proprietary
Wisent	[89]	LALR(1)	?	C++, Java	mixed	external	Java Virtual Machine	No	GNU GPL
Yacc (AT&T)/Sun	[90]/[91]	LALR(1)	YACC	C	mixed	external	POSIX	No	CPL & CDDL
Yacc++	[92]	LR(1), LALR(1)	YACC	C++, C#	mixed	generated or external	all	No	Proprietary
Yapps	[93]	LL(1)	?	Python	mixed	generated	all	No	MIT
yecc	[94]	LALR(1)	?	Erlang	separate	generated	all	No	Erlang
Visual BNF	[95]	LR(1), LALR(1)	?	C#	separate	generated	.NET Framework	Yes	Proprietary
Visual Parse++	[96]	LALR(???)	?	C, C++, C#, Java	separate	generated	Microsoft Windows	Yes	Proprietary
YooParse	[97]	LR(1), LALR(1)	?	C++	mixed	external	all	No	MIT
Product	Website	Parsing algorithm	Input grammar notation	Output languages	Grammar, code	Lexer	Development platform	IDE	License

Parsing expression grammars, deterministic boolean grammars

Name	Website	Parsing algorithm	Output languages	Grammar, code	Development platform	License
Aurochs	[98]	Packrat	C, Objective Caml, Java	mixed	all	GNU GPL
CL-peg	[99]	Packrat	Common Lisp	mixed	all	MIT
Drat!	[100]	Packrat	D	mixed	all	GNU GPL
Frisby	[101]	Packrat	Haskell	mixed	all	BSD
grammar::peg	[102]	Packrat	Tcl	mixed	all	BSD
IronMeta	[103]	Packrat	C#	mixed	Microsoft Windows	BSD
Katahdin	[104]	Packrat (modified), mutating interpreter	C#	mixed	all	Public domain
Laja	[4]	2-phase scannerless top-down backtracking + runtime support	Java	separate	all	GNU GPL
LPEG	[105]	Parsing Machine	Lua	mixed	all	MIT
Mouse	[106]	Recursive descent	Java	separate	Java Virtual Machine	Apache License 2.0
Narwhal	[107]	Packrat	C	mixed	POSIX, Microsoft Windows	BSD
Nemerle.Peg	[108]	Recursive descent + Pratt	Nemerle	separate	all	BSD
neotoma	[109]	Packrat	Erlang	separate	all	MIT
NPEG	[110]	Recursive descent	C#	mixed	all	MIT
OMeta	[111]	Packrat (modified)	JavaScript, Squeak	mixed	all	MIT
Packrat	[112]	Packrat	Scheme	mixed	all	MIT
Pappy	[113]	Packrat	Haskell	mixed	all	Proprietary
parboiled	[114]	Recursive descent	Java, Scala	mixed	Java Virtual Machine	Apache License 2.0
parsepp	[115]	?	C++	mixed	all	Public domain
Parsnip	[116]	Packrat	C++	mixed	Microsoft Windows	GNU GPL
peg	[117]	Recursive descent	C	mixed	all	MIT
PEG.js	[118]	Packrat	JavaScript	mixed	all	MIT
peg-parser	[119]	PEG parser interpreter	Dylan	separate	all	
pegc	[120]	?	C	mixed	all	Public domain
PetitParser	[121]	Packrat	Smalltalk, JavaScript	mixed	all	MIT
PEGTL	[122]	Recursive descent	C++0x	mixed	POSIX	MIT

PGE	Parser Grammar Engine	Recursive descent	Parrot bytecode	mixed	Parrot virtual machine	Artistic 2.0
PyPy rlib	[123]	Packrat	Python	mixed	all	MIT
pyPEG	[124]	PEG parser interpreter, Packrat	Python	mixed	all	GNU GPL
Rats!	[125]	Packrat	Java	mixed	Java Virtual Machine	GNU LGPL
Spirit2	[126]	Recursive descent	C++	mixed	all	Boost
Treetop	[127]	Recursive descent	Ruby	mixed	all	
Yard	[128]	Recursive descent	C++	mixed	all	MIT or Public domain
Waxeye	[129]	Packrat	C, Java, JavaScript, Python, Ruby, Scheme	separate	all	MIT

General context-free, conjunctive or boolean languages

Name	Website	Parsing algorithm	Input grammar notation	Output languages	Grammar, code	Lexer	Development platform	IDE	License
ACCENT	[130]	Earley	?	C	mixed	external	all	No	GNU GPL
APaGeD	[131]	GLR, LALR(1), LL(k)	?	D	mixed	generated	all	No	Artistic
Bison	[132]	LALR, GLR	YACC	C, C++, Java	mixed	external	all	No	GNU GPL
DMS Software Reengineering Toolkit	[133]	GLR	?	Parlanse	mixed	generated	Microsoft Windows	No	Proprietary
DParser	[3]	Scannerless GLR	?	C	mixed	scannerless	POSIX	No	BSD
Dypgen	[134]	runtime-extensible GLR	?	Objective Caml	mixed	generated	all	No	CeCILL-B
Elkhound	[135]	GLR	?	C++, Objective Caml	mixed	external	all	No	BSD
eu.h8me.Parsing	[136]	GLR	?	N/A (state machine is runtime generated)	separate	external	.NET Framework	No	BSD
GDK	[137]	LALR(1), GLR	?	C, Lex, Haskell, HTML, Java, Object Pascal, Yacc	mixed	generated	POSIX	No	MIT

Happy	[138]	LALR, GLR	?	Haskell	mixed	external	all	No	BSD
Hime Parser Generator	[43]	GLR	?	C#	separate	generated	.NET Framework	No	GNU LGPL
Jison	[139]	LALR(1), LR(0), SLR(1)	YACC	JavaScript	mixed	generated	all	No	MIT
Laja	[4]	Scannerless, two phase	Laja	Java	separate	scannerless	all	No	GNU GPL
Scannerless Boolean Parser	[140]	Scannerless GLR (Boolean grammars)	?	Haskell, Java	separate	scannerless	Java Virtual Machine	No	BSD
SDF/SGLR	[141]	Scannerless GLR	SDF	C, Java	separate	scannerless	all	Yes	BSD
SmaCC	[142]	GLR(1), LALR(1), LR(1)	?	Smalltalk	mixed	internal	all	Yes	MIT
SPARK	[14]	Earley	?	Python	mixed	external	all	No	MIT
Tom	[143]	GLR	?	C	generated	none	all	No	Proprietary
UltraGram	[144]	LALR, LR, GLR	?	C++, C#, Java, Visual Basic .NET	separate	generated	Microsoft Windows	Yes	Proprietary
Wormhole	[145]	Pruning, LR, GLR, Scannerless GLR	?	C, Python	mixed	scannerless	Microsoft Windows	No	MIT
Whale Calf	[146]	General tabular, SLL(k), Linear normal form (Conjunctive grammars), LR, Binary normal form (Boolean grammars)	?	C++	separate	external	all	No	Proprietary

External links

- The Catalog of Compiler Construction Tools [147]
- Python Language Parsing [148]
- Open Source Parser Generators in Java [149]

References

- [1] <http://languageengineering.biz/dfastar.html>
- [2] <http://users.utu.fi/aleokh/dolphin/>
- [3] <http://flex.sourceforge.net/>
- [4] <http://www.haskell.org/alex/>
- [5] <http://jflex.de/>
- [6] <http://sourceforge.net/projects/csflex>
- [7] <http://www.cs.princeton.edu/~appel/modern/java/JLex/>
- [8] <http://www.cs.princeton.edu/~appel/modern/java/JLex/#LICENSE>
- [9] <http://www.infosys.tuwien.ac.at/cuplex/lex.htm>
- [10] <http://code.google.com/p/cookcc/>
- [11] <http://www.gobosoft.com/eiffel/gobo/gelex/>
- [12] <http://gplex.codeplex.com/>
- [13] <http://quex.sourceforge.net/>
- [14] <http://www.complang.org/ragel/>

- [15] <http://re2c.org/>
- [16] <http://swtch.com/~rsc/regexp/>
- [17] <http://www.benhanson.net/lexertl.html>
- [18] <http://antlr.org/>
- [19] <http://coasttocoastresearch.com/>
- [20] <http://www.gbresearch.com/axe/Default.aspx>
- [21] <http://beaver.sourceforge.net/>
- [22] <ftp://ftp.iecc.com/pub/file/bison++flex++/>
- [23] <http://bisoncpp.sourceforge.net/>
- [24] <http://siber.com/btyacc/>
- [25] <http://invisible-island.net/byacc/byacc.html>
- [26] <http://byaccj.sourceforge.net/>
- [27] <http://www.pps.jussieu.fr/~jch/software/cl-yacc/>
- [28] <http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/>
- [29] <http://cppcc.sourceforge.net/>
- [30] <http://csparser.sourceforge.net/>
- [31] <http://cis.paisley.ac.uk/crow-ci0/>
- [32] <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
- [33] http://www.lemke-it.com/lit_opensource.html
- [34] <http://eli-project.sourceforge.net/>
- [35] <http://www.informatik.uni-freiburg.de/proglang/software/essence/>
- [36] <http://search.cpan.org/~casiano/Parse-Eyapp/>
- [37] <http://www.informatik.uni-bonn.de/~ralf/frown/>
- [38] <http://www.gobosoft.com/eiffel/gobo/geyacc/>
- [39] <http://devincook.com/goldparser/>
- [40] <http://gppg.codeplex.com/>
- [41] <http://grammatica.percederberg.net/>
- [42] <http://www.hilexed.org/>
- [43] <http://himeparser.codeplex.com/>
- [44] <http://hyacc.sourceforge.net/>
- [45] <http://web.cecs.pdx.edu/~mpj/jacc/index.html>
- [46] <https://javacc.dev.java.net/>
- [47] <http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html>
- [48] <http://www.jflap.org/>
- [49] <http://jetpag.sourceforge.net/>
- [50] <http://www.kdevelop.org/>
- [51] <http://www.complang.org/kelbt/>
- [52] <http://www005.upp.so-net.ne.jp/kmori/kmyacc/>
- [53] <http://sourceforge.net/projects/lapg/>
- [54] <http://hwaci.com/sw/lemon>
- [55] <http://www.acooke.org/lepl/>
- [56] <http://sourceforge.net/projects/lime-php/>
- [57] <http://labraj.uni-mb.si/lisa>
- [58] <http://sourceforge.net/projects/lpg/>
- [59] <http://freshmeat.net/projects/llgen/>
- [60] <http://os.ghalkes.nl/LLnextgen/>
- [61] <http://languageengineering.biz/lrstar.html>
- [62] <http://cristal.inria.fr/~fpottier/menhir/>
- [63] <http://christophe.delord.free.fr/mp/index.html>
- [64] <http://www.smlnj.org/doc/ML-Yacc/>
- [65] <http://sourceforge.net/projects/monkey/>
- [66] <http://babel.ls.fi.upm.es/research/mtp/>
- [67] <http://cocom.sourceforge.net/>
- [68] <http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html>
- [69] <http://www.irule.be/bvh/c++/olex/>
- [70] <http://legacy.cs.uu.nl/daan/parsec.html>
- [71] <http://freshmeat.net/projects/parse-yapp/>
- [72] <http://www.polhode.com/pccts.html>
- [73] <http://www.dabeaz.com/ply/>

- [74] <http://vl.fmnet.info/precc/>
- [75] <http://labs.trolltech.com/page/Projects/Compilers/QLALR>
- [76] <http://rpatk.net>
- [77] <http://sablecc.org/>
- [78] <http://members.cox.net/slkgpg/>
- [79] <http://spirit.sourceforge.net/>
- [80] <http://www.speculate.de/>
- [81] http://www.sweetsoftware.co.nz/parser_overview.html
- [82] <http://tap.sourceforge.net/>
- [83] <http://textransformer.com/>
- [84] <http://www.codeproject.com/KB/recipes/TinyPG.aspx>
- [85] <http://www.codeproject.com/info/cpol10.aspx>
- [86] <http://christophe.delord.free.fr/tpg/>
- [87] <http://www.musikwissenschaft.uni-mainz.de/~ag/tply/tply.html>
- [88] <http://users.utu.fi/aleokh/whale/>
- [89] <http://sourceforge.net/projects/wisent/>
- [90] <http://www.research.att.com/sw/tools/uwin>
- [91] <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/sgs/yacc/>
- [92] <http://world.std.com/~compres/>
- [93] <http://theory.stanford.edu/~amitp/yapps/>
- [94] <http://www.erlang.org/faq/parsing.html>
- [95] <http://intralogic.eu/>
- [96] <http://visual-parse.software.informer.com/>
- [97] <http://yooparse.sourceforge.net/>
- [98] <http://aurochs.fr>
- [99] <http://common-lisp.net/project/cl-peg/>
- [100] <http://wiki.dprogramming.com/Drat/HomePage/>
- [101] <http://repetae.net/computer/frisby/>
- [102] <http://tcllib.sourceforge.net/doc/peg.html>
- [103] <http://ironmeta.sourceforge.net/>
- [104] <http://www.chrisseaton.com/katahdin/>
- [105] <http://www.inf.puc-rio.br/~roberto/lpeg/>
- [106] <http://mousepeg.sourceforge.net/>
- [107] <http://sourceforge.net/projects/narwhal/>
- [108] <http://nemerle.googlecode.com/svn/nemerle/trunk/snippets/peg-parser>
- [109] <http://github.com/seancribbs/neotoma>
- [110] <http://www.codeplex.com/NPEG>
- [111] <http://tinlizzie.org/ometa/>
- [112] <http://www.call-with-current-continuation.org/eggs/packrat.html>
- [113] <http://pdos.csail.mit.edu/~baford/packrat/thesis/>
- [114] <http://parboiled.org>
- [115] <http://fossil.wanderinghorse.net/repos/parsepp/index.cgi/wiki/parsepp>
- [116] <http://parsnip-parser.sourceforge.net/>
- [117] <http://piumarta.com/software/peg/>
- [118] <http://pegjs.majda.cz/>
- [119] <http://wiki.opendylan.org/wiki/view.dsp?title=PEG%20Parser%20Library>
- [120] <http://fossil.wanderinghorse.net/repos/pegc/index.cgi/index>
- [121] <http://source.lukas-renggli.ch/petit.html>
- [122] <http://code.google.com/p/pegtl/>
- [123] <http://codespeak.net/pypy/dist/pypy/doc/rlib.html#parsing>
- [124] <http://fdik.org/pyPEG>
- [125] <http://cs.nyu.edu/rgrimm/xtc/rats.html>
- [126] <http://www.boost.org/>
- [127] <http://treetop.rubyforge.org/>
- [128] <http://code.google.com/p/yardparser/>
- [129] <http://waxeye.org>
- [130] <http://accent.compilertools.net/>
- [131] <http://apaged.mainia.de/>
- [132] <http://gnu.org/software/bison/>

[133] <http://semanticdesigns.com/Products/DMS/DMSToolkit.html>
[134] <http://dypgen.free.fr/>
[135] <http://scottmcpeak.com/elkhound/>
[136] <http://parsing.codeplex.com>
[137] <http://sourceforge.net/projects/gdk>
[138] <http://www.haskell.org/happy/>
[139] <http://zaach.github.com/json>
[140] <http://research.cs.berkeley.edu/project/sbp/>
[141] <http://www.syntax-definition.org/>
[142] <http://refactory.com/Software/SmaCC/>
[143] <http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/nlp/parsing/tom/0.html>
[144] <http://www.ultragram.com>
[145] <http://www.mightyheave.com>
[146] <http://users.utu.fi/aleokh/whalecalf/>
[147] <http://catalog.compilertools.net/lexparse.html>
[148] <http://wiki.python.org/moin/LanguageParsing>
[149] <http://java-source.net/open-source/parser-generators>

Lex

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").^[1] ^[2] Lex is commonly used with the yacc parser generator. Lex, originally written by Mike Lesk and Eric Schmidt,^[3] is the standard lexical analyzer generator on many Unix systems, and a tool exhibiting its behavior is specified as part of the POSIX standard.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

Though traditionally proprietary software, versions of Lex based on the original AT&T code are available as open source, as part of systems such as OpenSolaris and Plan 9 from Bell Labs. Another popular open source version of Lex is flex, the "fast lexical analyzer".

Structure of a lex file

The structure of a lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

- The **definition** section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Example of a lex file

The following is an example lex file for the flex version of lex. It recognizes strings of numbers (integers) in the input, and simply prints them out.

```
/* *** Definition section *** /  
  
%{  
/* C code to be copied verbatim */  
#include <stdio.h>  
%}  
  
/* This tells flex to read only one input file */  
%option noyywrap  
  
%%  
/* *** Rules section *** /  
  
/* [0-9]+ matches a string of one or more digits */  
[0-9]+ {  
    /* yytext is a string containing the matched text. */  
    printf("Saw an integer: %s\n", yytext);  
}  
  
.|\n      { /* Ignore all other characters. */ }  
  
%%  
/* *** C Code section *** /  
  
int main(void)  
{  
    /* Call the lexer, then quit. */  
    yylex();  
    return 0;  
}
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input:

```
abc123z.!&*2gj6
```

the program will print:

```
Saw an integer: 123  
Saw an integer: 2  
Saw an integer: 6
```

Using Lex with other programming tools

Using Lex with parser generators

Lex and parser generators, such as Yacc or Bison, are commonly used together. Parser generators use a formal grammar to parse an input stream, something which Lex cannot do using simple regular expressions (Lex is limited to simple finite state automata).

It is typically preferable to have a (Yacc-generated, say) parser be fed a token-stream as input, rather than having it consume the input character-stream directly. Lex is often used to produce such a token-stream.

Scannerless parsing refers to where a parser consumes the input character-stream directly, without a distinct lexer.

Lex and make

make is a utility that can be used to maintain programs involving lex. Make assumes that a file that has an extension of `.l` is a lex source file. The make internal macro `LFLAGS` can be used to specify lex options to be invoked automatically by make.^[4]

References

- [1] Levine, John R; Mason, Tony; Brown, Doug (1992). *LEX & YACC* (<http://books.google.co.in/books?id=YrzpxNYegEkC&printsec=frontcover#PPA1,M1>) (2 ed.). O'Reilly. pp. 1–2. ISBN 1-56592-000-7. .
- [2] Levine, John (August 2009). *flex & bison* (<http://oreilly.com/catalog/9780596155988>). O'Reilly Media. pp. 304. ISBN 978-0-596-15597-1. .
- [3] Schmidt, Eric. "Lex - A Lexical Analyzer Generator" (<http://dinosaur.compilers.net/lex/index.html>). . Retrieved 16 August 2010.
- [4] "make" (<http://www.opengroup.org/onlinepubs/009695399/utilities/make.html>), *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition* (The IEEE and The Open Group), 2004,

External links

- Using Flex and Bison at Macworld.com (<http://www.mactech.com/articles/mactech/Vol.16/16.07/UsingFlexandBison/>)

flex lexical analyser

flex

Developer(s)	Vern Paxson
Stable release	2.5.35 / February 26, 2008
Operating system	Unix-like
Type	Lexical analyzer generator
License	BSD license
Website	http://flex.sourceforge.net/

flex (fast lexical analyzer generator) is a free software alternative to lex.^[1] It is frequently used with the free Bison parser generator. Unlike Bison, flex is not part of the GNU Project.^[2] Flex was written in C by Vern Paxson around 1987. He was translating a Ratfor generator, which had been led by Jef Poskanzer.^[3]

A similar lexical scanner for C++ is flex++, which is included as part of the flex package. At the moment, flex supports generating code only for C and C++ (see flex++). The generated code does not depend on any runtime or external library except for a memory allocator (malloc or a user-supplied alternative) unless the input also depends on it. This can be useful in embedded and similar situations where traditional operating system or C runtime facilities may not be available.

Example lexical analyzer

This is an example of a scanner which does not make use of Flex (written in C) for the instructional programming language PL/0.

The symbols recognized are: '+', '-', '*', '/', '=', '(', ')', ',', ';', '.', ':='; '<', '<='; '>', '>='; numbers: 0–9 {0–9}; identifiers: a-zA-Z {a-zA-Z0–9} and keywords: begin, call, const, do, end, if, odd, procedure, then, var, while.

External variables used:

```
FILE *source                                /* The source file */
int cur_line, cur_col, err_line, err_col /* For error reporting */
int num                                     /* Last number read stored
here, for the parser */
char id[]                                    /* Last identifier read stored
here, for the parser */
Hashtab *keywords                            /* List of keywords */
```

External routines called:

```
error(const char msg[])                      /* Report an error */
*/
Hashtab *create_htab(int estimate)           /* Create a lookup
table */
int enter_htab(Hashtab *ht, char name[], void *data) /* Add an entry to
a lookup table */
Entry *find_htab(Hashtab *ht, char *s)        /* Find an entry
in a lookup table */
```

```

void *get_htab_data(Entry *entry)          /* Returns data
from a lookup table */
FILE *fopen(char fn[], char mode[])        /* Opens a file
for reading */
fgetc(FILE *stream)                       /* Read the next
character from a stream */
ungetc(int ch, FILE *stream)               /* Put-back a
character onto a stream */
isdigit(int ch), isalpha(int ch), isalnum(int ch)    /* Character
classification */

```

External types:

```

Symbol /* An enumerated type of all the symbols in the PL/0 language */
Hashtab /* Represents a lookup table */
Entry /* Represents an entry in the lookup table */

```

Scanning is started by calling `init_scan`, passing the name of the source file. If the source file is successfully opened, the parser calls `getsym` repeatedly to return successive symbols from the source file.

The heart of the scanner, `getsym`, should be straightforward. First, whitespace is skipped. Then the retrieved character is classified. If the character represents a multiple-character symbol, additional processing must be done. Numbers are converted to internal form, and identifiers are checked to see if they represent a keyword.

```

int read_ch(void) {
    int ch = fgetc(source);
    cur_col++;
    if (ch == '\n') {
        cur_line++;
        cur_col = 0;
    }
    return ch;
}

void put_back(int ch) {
    ungetc(ch, source);
    cur_col--;
    if (ch == '\n') cur_line--;
}

Symbol getsym(void) {
    int ch;

    while ((ch = read_ch()) != EOF && ch <= ' ')
        ;
    err_line = cur_line;
    err_col = cur_col;
    switch (ch) {
        case EOF: return eof;
        case '+': return plus;
    }
}

```

```
case '-': return minus;
case '*': return times;
case '/': return slash;
case '=': return eql;
case '(': return lparen;
case ')': return rparen;
case ',': return comma;
case ';': return semicolon;
case '.': return period;
case ':':
    ch = read_ch();
    return (ch == '=')? becomes : nul;
case '<':
    ch = read_ch();
    if (ch == '>') return neq;
    if (ch == '=') return leq;
    put_back(ch);
    return lss;
case '>':
    ch = read_ch();
    if (ch == '=') return geq;
    put_back(ch);
    return gtr;
default:
    if (isdigit(ch)) {
        num = 0;
        do { /* no checking for overflow! */
            num = 10 * num + ch - '0';
            ch = read_ch();
        } while (ch != EOF && isdigit(ch));
        put_back(ch);
        return number;
    }
    if (isalpha(ch)) {
        Entry *entry;
        id_len = 0;
        do {
            if (id_len < MAX_ID) {
                id[id_len] = (char)ch;
                id_len++;
            }
            ch = read_ch();
        } while (ch != EOF && isalnum(ch));
        id[id_len] = '\0';
        put_back(ch);
        entry = find_htab(keywords, id);
        return entry ? (Symbol)get_htab_data(entry) : ident;
```

```

    }

    error("getsym: invalid character '%c'", ch);
    return nul;
}

int init_scan(const char fn[]) {
    if ((source = fopen(fn, "r")) == NULL) return 0;
    cur_line = 1;
    cur_col = 0;
    keywords = create_htab(11);
    enter_htab(keywords, "begin", beginsym);
    enter_htab(keywords, "call", callsym);
    enter_htab(keywords, "const", constsym);
    enter_htab(keywords, "do", dosym);
    enter_htab(keywords, "end", endsym);
    enter_htab(keywords, "if", ifsym);
    enter_htab(keywords, "odd", oddsym);
    enter_htab(keywords, "procedure", procsym);
    enter_htab(keywords, "then", thensym);
    enter_htab(keywords, "var", varsym);
    enter_htab(keywords, "while", whilesym);
    return 1;
}

```

Now, contrast the above code with the code needed for a flex generated scanner for the same language:

```
%{
#include "y.tab.h"
%}

digit      [0-9]
letter     [a-zA-Z]

%%
"+"
"-"
"*"
"/"
"("
")"
";"
","
"."
":="
"="
"<>"

{ return PLUS; }
{ return MINUS; }
{ return TIMES; }
{ return SLASH; }
{ return LPAREN; }
{ return RPAREN; }
{ return SEMICOLON; }
{ return COMMA; }
{ return PERIOD; }
{ return BECOMES; }
{ return EQL; }
{ return NEQ; }
```

```

"<"           { return LSS;      }
">"           { return GTR;      }
"<="          { return LEQ;      }
">="          { return GEQ;      }
"begin"        { return BEGINSYM; }
"call"         { return CALLSYM; }
"const"        { return CONSTSYM; }
"do"           { return DOSYM;    }
"end"          { return ENDSYM;   }
"if"           { return IFSYM;    }
"odd"          { return ODDSYM;   }
"procedure"    { return PROCSYM;  }
"then"         { return THENSYM;  }
"var"          { return VARSYM;   }
"while"        { return WHILESYM; }

{letter}({letter}|{digit})* {
    yyval.id = (char *) strdup(yytext);
    return IDENT;
}
{digit}+
/* skip whitespace */
.
{ printf("Unknown character [%c]\n", yytext[0]);
    return UNKNOWN;
}

%%

int yywrap(void){return 1;}

```

About 50 lines of code for flex versus about 100 lines of hand-written code.

Issues

Time complexity

A Flex lexical analyzer sometimes has time complexity $O(n)$ in the length of the input. That is, it performs a constant number of operations for each input symbol. This constant is quite low: GCC generates 12 instructions for the DFA match loop. Note that the constant is independent of the length of the token, the length of the regular expression and the size of the DFA.

However, one optional feature of Flex can cause Flex to generate a scanner with non-linear performance: The use of the REJECT macro in a scanner with the potential to match extremely long tokens. In this case, the programmer has explicitly told flex to "go back and try again" after it has already matched some input. This will cause the DFA to backtrack to find other accept states. In theory, the time complexity is $O(n + m^2) \geq O(m^2)$ where m is the length of the longest token (this reverts to $O(n)$ if tokens are "small" with respect to the input size).^[4] The REJECT feature is not enabled by default, and its performance implications are thoroughly documented in the Flex manual.

Reentrancy

By default the scanner generated by Flex is not reentrant. This can cause serious problems for programs that use the generated scanner from different threads. To overcome this issue there are options that Flex provides in order to achieve reentrancy. A detailed description of these options can be found in the Flex manual.^[5]

Usage under non-Unix environments

Normally the generated scanner contains references to *unistd.h* header file which is Unix specific. To avoid generating code that includes *unistd.h*, *%option nounistd* should be used. Another issue is the call to *isatty* (a Unix library function), which can be found in the generated code. The *%option never-interactive* forces flex to generate code that doesn't use *isatty*. These options are detailed in the Flex manual.^[6]

Using flex from other languages

Flex can only generate code for C and C++. To use the scanner code generated by flex from other languages a language binding tool such as SWIG can be used.

Flex++

Flex++ is a tool for creating a language parsing program. A parser generator creates a language parsing program. It is a general instantiation of the flex program.

These programs perform character parsing, and tokenizing via the use of a deterministic finite automata or Deterministic finite state machine (DFA). A DFA (or NDFA) is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read only right moving Turing Machines or NDFA. The syntax is based on the use of Regular expressions.

Flex provides two different ways to generate scanners. It primarily generates C code to be compiled as opposed to C++ libraries and code. Flex++, an extension of flex, is used for generating C++ code and classes. The Flex++ classes and code require a C++ compiler to create lexical and pattern-matching programs. Flex, the alternative language parser, defaults to generating a parsing scanner in C code. The Flex++ generated C++ scanner includes the header file FlexLexer.h, which defines the interfaces of the two C++ generated classes.

Flex creators

Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson.

References

- [1] Levine, John (August 2009). *flex & bison* (<http://oreilly.com/catalog/9780596155988>). O'Reilly Media. pp. 304. ISBN 978-0-596-15597-1.
- [2] Is flex GNU or not? (http://flex.sourceforge.net/manual/Is-flex-GNU-or-not_003f.html#Is-flex-GNU-or-not_003f), flex FAQ
- [3] When was flex born? (http://flex.sourceforge.net/manual/When-was-flex-born_003f.html#When-was-flex-born_003f), flex FAQ
- [4] <http://flex.sourceforge.net/manual/Performance.html> (last paragraph)
- [5] <http://flex.sourceforge.net/manual/Reentrant.html>
- [6] http://flex.sourceforge.net/manual/Code_002dLevel-And-API-Options.html

External links

- Flex Homepage (<http://flex.sourceforge.net/>)
- Flex Manual (<http://flex.sourceforge.net/manual/>)
- ANSI-C Lex Specification (<http://www.quut.com/c/ANSI-C-grammar-l-1998.html>)
- JFlex: Fast Scanner Generator for Java (<http://www.jflex.de/>)
- Brief description of Lex, Flex, YACC, and Bison (<http://dinosaur.compilertools.net/>)
- Compiler Construction using Flex and Bison - course by Anthony Aaby (<http://www.cs.wwc.edu/~aabyan/464/Book/index.html>) A Romanian rebuild version of this book in pdf format "fb2-press.pdf" or "fb2-printing.pdf" may be downloaded from (<http://cs.wwc.edu/~aabyan/publications.html>) or directly from (<http://cs.wwc.edu/~aabyan/Linux/fb2-press.pdf>) or (<http://cs.wwc.edu/~aabyan/Linux/fb2-printing.pdf>).
- Download Win32 binaries of Flex++ and Bison++ (<http://www.kohsuke.org/flex++bison++/>)

Quex

For the Nazi propaganda movie, see Hitler Youth Quex.

quex

Developer(s)	Dr.-Ing. Frank-Rene Schäfer
Stable release	0.59.6 / July 14, 2011
Operating system	Cross-platform
Type	Lexical analyzer generator
License	LGPL
Website	quex.sourceforge.net [13]

Quex is a lexical analyzer generator that creates C and C++ lexical analyzers. Significant features include the ability to generate lexical analyzers that operate on Unicode input, the creation of direct coded (non-table based) lexical analyzers and the use of inheritance relationships in lexical analysis modes.

Features

Direct coded lexical analyzers

Quex uses traditional steps of Thompson construction to create nondeterministic finite-state machines from regular expressions, conversion to a deterministic finite-state machine and then Hopcroft optimization to reduce the number of states to a minimum. Those mechanisms, though, have been adapted to deal with character sets rather than single characters. By means of this the calculation time can be significantly reduced. Since the Unicode character set consists of many more code points than plain ASCII, those optimizations are necessary in order to produce lexical analysers in a reasonable amount of time.

Instead of construction of a table based lexical analyzer where the transition information is stored in a data structure, Quex generates C/C++ code to perform transitions. Direct coding creates lexical analyzers that structurally more closely resemble typical hand written lexical analyzers than table based lexers. Also direct coded lexers tend to perform better than analogous table based lexical analyzers.

Unicode input alphabets

Quex can handle input alphabets that contain the full Unicode code point range (0 to 10FFFFh). This is augmented by the ability to specify regular expressions that contain Unicode properties as expressions. For example, Unicode code points with the binary property XID_Start can be specified with the expression \P{XID_Start} or \P{XIDS}. Quex can also generate code to call iconv or ICU to perform character conversion. Quex relies directly on databases as they are delivered by the Unicode Consortium. Updating to new releases of the standard consists only of copying the correspondent database files into Quex's corresponding directory.

Lexical analysis modes

Like traditional lexical analyzers (e.g. Lex and Flex), Quex supports multiple lexical analysis modes in a lexer. In addition to pattern actions, Quex modes can specify event actions: code to be executed during events such as entering or exiting a mode or when any match is found. Quex modes can be also be related by inheritance which allows modes to share common pattern and event actions.

Sophisticated buffer handling

Quex provides sophisticated mechanism of buffer handling and reload that are at the same time efficient and flexible. Quex provides interfaces that allow users to virtually plug-in any character set converter. The converters are activated only "on-demand", that is, when new buffer filling is required. By default Quex can plug-in the iconv library. By means of this backbone Quex is able to analyze a huge set of character encodings.

Example

Quex follows the syntax of the classical tools lex and flex for the description of regular expressions. The example in the section Flex can be translated into Quex source code as follows:

```
header {
    #include <cstdlib> // C++ version of 'stdlib.h'
}

define {
    digit      [0-9]
    letter     [a-zA-Z]
}

mode X :
<skip:    [ \t\n\r]>
{
    "+"          => QUEX_TKN_PLUS;
    "-"          => QUEX_TKN_MINUS;
    "*"          => QUEX_TKN_TIMES;
    "/"          => QUEX_TKN_SLASH;
    "("          => QUEX_TKN_LPAREN;
    ")"          => QUEX_TKN_RPAREN;
    ";"          => QUEX_TKN_SEMICOLON;
    ","          => QUEX_TKN_COMMA;
    "."          => QUEX_TKN_PERIOD;
    ":"          => QUEX_TKN_BECOMES;
    "="          => QUEX_TKN_EQ;
    "<>"        => QUEX_TKN_NEQ;
    "<"          => QUEX_TKN_LSS;
    ">"          => QUEX_TKN_GTR;
    "<="         => QUEX_TKN_LEQ;
    ">="         => QUEX_TKN_GEQ;
    "begin"      => QUEX_TKN_BEGINSYM;
    "call"       => QUEX_TKN_CALLSYM;
    "const"      => QUEX_TKN_CONSTSYM;
```

```

"do"                  => QUEX_TKN_DOSYM;
"end"                => QUEX_TKN_ENDSYM;
"if"                 => QUEX_TKN_IFSYM;
"odd"                => QUEX_TKN_ODDSYM;
"procedure"          => QUEX_TKN_PROCSYM;
"then"               => QUEX_TKN_THENSYM;
"var"                => QUEX_TKN_VARSYM;
"while"              => QUEX_TKN_WHILESYM;

{letter} ({letter} | {digit})* => QUEX_TKN_IDENT ( strdup (Lexeme) );
{digit}+
.
.
.

}

```

The brief token senders via the "`=>`" operator set the token ID of a token object with the token ID which follows the operator. The arguments following inside brackets are used to set contents of the token object. Note, that skipping whitespace can be achieved via skippers which are optimized to pass specific character sets quickly (see the "`<skip: ...>`" tag). For more sophisticated token actions C-code sections can be provided, such as

```

...
{digit}+      {
    if( is_prime_number(Lexeme) )      ++prime_number_counter;
    if( is_fibonacci_number(Lexeme) )
++fibonacci_number_counter;
    self.send(QUEX_TKN_NUMBER(atoi(Lexeme)));
}
...

```

which might be used to do some statistics about the numbers which occur in analyzed code.

External links

- Quex ^[13], official website
- Lexical Analyzer Generator Quex ^[1] at SourceForge.net

References

[1] <http://sourceforge.net/projects/quex/>

JLex

JLex is a lexical analyser (similar to Lex) which produces Java language source code. It is implemented in Java. JLex was developed by Elliot Berk at Princeton University.

External links

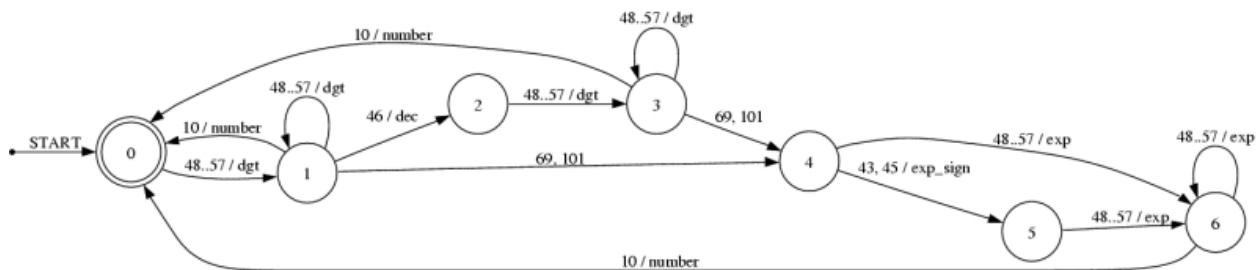
- Main website with source code and documentation ^[7]

Ragel

Ragel

Developer(s)	Adrian Thurston
Stable release	6.7 / 2011-05-22
Operating system	Unix-like, Windows
Type	State machine compiler
License	GNU General Public License
Website	www.complang.org/ragel/ [14]

Ragel is a finite-state machine compiler with output support for C, C++, C#, Objective-C, D, Java, Go and Ruby source code. It supports the generation of table or control flow driven state machines from regular expressions and/or state charts and can also build lexical analysers via the longest-match method. A unique feature of Ragel is that user actions can be associated with arbitrary state machine transitions using operators that are integrated into the regular expressions. Ragel also supports visualization of the generated machine via graphviz.



The above graph does the following: take a user input as a series of bytes. If in 48..57 (e.g. '0' to '9') it begins like a number. If 10 is encountered, we're done. 46 is the '.' and indicates a decimal number. 69/101 is uppercase/lowercase 'e' and indicates a number in scientific format. As such it will recognize properly:

2 45 055 46. 78.1 2e5 78.3e12 69.0e-3 3e+3

but not: .3 -5 3.e2 2e5.1

External links

- Ragel website [14]
- Ragel user guide [1]
- Ragel mailing list [2]

References

- [1] <http://www.complang.org/ragel/ragel-guide-6.6.pdf>
[2] <http://www.complang.org/mailman/listinfo/ragel-users>

yacc

The computer program **yacc** is a parser generator developed by Stephen C. Johnson at AT&T for the Unix operating system. The name is an acronym for "Yet Another Compiler Compiler." It generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to BNF.

Yacc used to be available as the default parser generator on most Unix systems. It has since been supplanted as the default by more recent, largely compatible, programs such as Berkeley Yacc, GNU bison, MKS yacc and Abraxas pcyacc. An updated version of the original AT&T version is included as part of Sun's OpenSolaris project. Each offers slight improvements and additional features over the original yacc, but the concept has remained the same. Yacc has also been rewritten for other languages, including Ratfor, ML, Ada, Pascal, Java, Python, Ruby and Common Lisp.

The parser generated by yacc requires a lexical analyzer. Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and Yacc.

Some versions of AT&T Yacc have become open source. For example, source code (for different implementations) is available with the standard distributions of Plan 9 and OpenSolaris.

References

- Stephen C. Johnson. YACC: Yet Another Compiler-Compiler^[1]. *Unix Programmer's Manual* Vol 2b, 1979.

External links

- Computerworld Interview with Stephen C. Johnson on YACC^[2]
- ML-Yacc^[3] a yacc version for the Standard ML language.
- CL-Yacc^[27], a LALR(1)parser generator for Common Lisp.
- PLY^[73] a yacc version for Python
- Yacc theory^[4]
- ocamlyacc^[68] a yacc version for Objective Caml.
- Racc^[5] a yacc version for Ruby.
- Paper "Parsing Non-LR(k) Grammars with Yacc"^[6] by Gary H. Merrill
- ANSI C yacc grammar^[7]

References

- [1] <http://dinosaur.compiletools.net/yacc/>
- [2] http://www.techworld.com.au/article/252319/-z_programming_languages_yacc
- [3] <http://www.smlnj.org/doc/ML-Yacc/index.html>
- [4] <http://epaperpress.com/lexandyacc/>
- [5] <http://github.com/tenderlove/racc>
- [6] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.1958>
- [7] <http://www.quut.com/c/ANSI-C-grammar-y.html>

Berkeley Yacc

Berkeley Yacc is a reimplementation of the Unix parser generator Yacc, originally written by Robert Corbett in 1990, designed for compatibility with Yacc.^[1] Due to its liberal license and because it was faster than the AT&T Yacc, it quickly became the most popular version of Yacc.^[2] It has the advantages of being written in ANSI C and being public domain software.

References

- [1] Doug Brown; John Levine; Tony Mason (October 1992), *lex & yacc* (2 ed.), O'Reilly Media
- [2] John Levine (August 2009), *flex & bison*, O'Reilly Media

External links

- Project home page for a ANSI C version (<http://invisible-island.net/byacc/byacc.html>)
- 1993 latest release by Berkeley (<ftp://ftp.cs.berkeley.edu/pub/4bsd/byacc.1.9.tar.Z>) sha1 sum-8e8138cd8c81365447c518c03830a59282b47a6e

ANTLR

ANTLR

Original author(s)	Terence Parr and others
Initial release	February 1992
Stable release	3.4 / July 18, 2011
Development status	in active development
Written in	Java
Platform	Cross-platform
License	BSD License
Website	antlr.org [1]

In computer-based language recognition, **ANTLR** (pronounced *Antler*), or *ANOther Tool for Language Recognition*, is a parser generator that uses LL(*) parsing. ANTLR is the successor to the **Purdue Compiler Construction Tool Set (PCCTS)**, first developed in 1989, and is under active development. Its maintainer is professor Terence Parr of the University of San Francisco.

ANTLR takes as input a grammar that specifies a language and generates as output source code for a recognizer for that language. At the moment, ANTLR supports generating code in the programming languages Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, and Ruby. A language is specified using a context-free grammar which is expressed using Extended Backus Naur Form EBNF.

ANTLR allows generating parsers, lexers, tree parsers, and combined lexer-parsers. Parsers can automatically generate abstract syntax trees which can be further processed with tree parsers. ANTLR provides a single consistent notation for specifying lexers, parsers, and tree parsers. This is in contrast with other parser/lexer generators and adds greatly to the tool's ease of use.

By default, ANTLR reads a grammar and generates a recognizer for the language defined by the grammar (i.e. a program that reads an input stream and generates an error if the input stream does not conform to the syntax specified by the grammar). If there are no syntax errors, then the default action is to simply exit without printing any message. In order to do something useful with the language, actions can be attached to grammar elements in the grammar. These actions are written in the programming language in which the recognizer is being generated. When the recognizer is being generated, the actions are embedded in the source code of the recognizer at the appropriate points. Actions can be used to build and check symbol tables and to emit instructions in a target language, in the case of a compiler.

As well as lexers and parsers, ANTLR can be used to generate tree parsers. These are recognizers that process abstract syntax trees which can be automatically generated by parsers. These tree parsers are unique to ANTLR and greatly simplify the processing of abstract syntax trees.

ANTLR 3 is free software, published under a three-clause BSD License. Prior versions were released as public domain software.^[2]

While ANTLR itself is free, however, the documentation necessary to use it is not. The ANTLR manual is a commercial book, *The Definitive ANTLR Reference*. Free documentation is limited to a handful of tutorials, code examples, and very basic API listings.

Several plugins have been developed for the Eclipse development environment to support the ANTLR grammar. There is ANTLR Studio, a proprietary product, as well as the ANTLR 2^[3] and 3^[4] plugins for Eclipse hosted on

SourceForge.

References

- [1] <http://www.antlr.org>
- [2] <http://www.antlr.org/pipermail/antlr-interest/2004-February/006340.html>
- [3] <http://antlreclipse.sourceforge.net/>
- [4] <http://antlrv3ide.sourceforge.net/>

Bibliography

- Parr, Terence (May 17, 2007), *The Definitive Antlr Reference: Building Domain-Specific Languages* (<http://www.pragprog.com/titles/tpantlr/the-definitive-antlr-reference>) (1st ed.), Pragmatic Bookshelf, pp. 376, ISBN 0978739256
- Parr, Terence (December, 2009), *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages* (<http://www.pragprog.com/titles/tpdsl/language-implementation-patterns>) (1st ed.), Pragmatic Bookshelf, pp. 374, ISBN 978-1-934356-45-6

External links

- Official website (<http://www.antlr.org>)
- ANTLRWorks (<http://www.antlr.org/works>)
- ANTLR Studio (<http://www.placidsystems.com/antlrstudio.aspx>)
- ANTLR For Delphi Target (<http://www.sharpplus.com/antlr-delphi-3-1>)
- ANTLR tutorial (<http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/antlr/antlr.html>) at the University of Birmingham
- For background on the theory, see articles from the ANTLR pages, e.g. T. J. Parr, R. W. Quong, *ANTLR: A Predicated-LL(k) Parser Generator* (<http://www.antlr.org/article/1055550346383/antlr.pdf>), *Software—Practice and Experience*, Vol. 25(7), 789–810 (July 1995)

GNU bison

GNU Bison

Developer(s)	The GNU Project
Stable release	(May 14, 2011) ^[1] ^[2] [+/-]
Operating system	Cross-platform
Type	Parser generator
License	GPL (free software)
Website	www.gnu.org/software/bison ^[3]

GNU bison, commonly known as Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) which reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison^[4] generates LALR parsers. Bison also supports “Generalized Left-to-right Rightmost” (GLR) parsers for grammars that are not LALR.

In POSIX mode, Bison is compatible with yacc, but also supports several improvements over this earlier program. flex, an automatic lexical analyser, is often used with Bison, to tokenise input data and provide Bison with tokens.

Bison is licensed as free software and is available in source code form. Earlier releases of bison used to stipulate that parts of its output were protected under the GPL, due to the inclusion of the yyparse() function from the original source code in the output. However, an exception was made, to allow other licenses to apply to the use of the output.^[5]

A complete reentrant parser example

The example shows how to use bison and flex to do a simple calculator (only add and multiply) and provides also an example for creating an abstract syntax tree. The next two files provide definition and implementation of the syntax tree functions.

```
/*
 * Expression.h
 * Definition of the structure used to build the syntax tree.
 */
#ifndef __EXPRESSION_H__
#define __EXPRESSION_H__

/**
 * @brief The operation type
 */
typedef enum tagEOperationType
{
    eVALUE,
    eMULTIPLY,
    ePLUS
} EOperationType;
```

```
/***
 * @brief The expression structure
 */
typedef struct tagSExpression
{
    EOperationType type; /////type of operation

    int value; /////valid only when type is eVALUE
    struct tagSExpression* left; /////left side of the tree
    struct tagSExpression* right; /////right side of the tree
} SExpression;

/***
 * @brief It creates an identifier
 * @param value The number value
 * @return The expression or NULL in case of no memory
 */
SExpression* createNumber(int value);

/***
 * @brief It creates an operation
 * @param type The operation type
 * @param left The left operand
 * @param right The right operand
 * @return The expression or NULL in case of no memory
 */
SExpression* createOperation(
    EOperationType type,
    SExpression *left,
    SExpression *right);

/***
 * @brief Deletes a expression
 * @param b The expression
 */
void deleteExpression(SExpression *b);

#endif // __EXPRESSION_H__

/*
 * Expression.c
 * Implementation of functions used to build the syntax tree.
 */

#include "Expression.h"

#include <stdlib.h>
```

```
/**  
 * @brief Allocates space for expression  
 * @return The expression or NULL if not enough memory  
 */  
static SExpression* allocateExpression()  
{  
    SExpression* b = malloc(sizeof *b);  
  
    if( b == NULL ) return NULL;  
  
    b->type = eVALUE;  
    b->value = 0;  
  
    b->left = NULL;  
    b->right = NULL;  
  
    return b;  
}  
  
SExpression* createNumber(int value)  
{  
    SExpression* b = allocateExpression();  
  
    if( b == NULL ) return NULL;  
  
    b->type = eVALUE;  
    b->value = value;  
  
    return b;  
}  
  
SExpression *createOperation(  
    EOperationType type,  
    SExpression *left,  
    SExpression *right)  
{  
    SExpression* b = allocateExpression();  
  
    if( b == NULL ) return NULL;  
  
    b->type = type;  
    b->left = left;  
    b->right = right;  
  
    return b;  
}
```

```

void deleteExpression(SExpression *b)
{
    if (b == NULL) return;

    deleteExpression(b->left);
    deleteExpression(b->right);

    free(b);
}

```

Since the tokens are provided by flex we must provide the means to communicate between the parser and the lexer. This is accomplished by defining the `YYSTYPE`. More details on this can be found on the flex manual.^[6]

```

/*
 * TypeParser.h
 * Definition of the structure used internally by the parser and lexer
 * to exchange data.
 */

#ifndef __TYPE_PARSER_H__
#define __TYPE_PARSER_H__


#include "Expression.h"

/***
 * @brief The structure used by flex and bison
 */
typedef union tagTypeParser
{
    SExpression *expression;
    int value;
} STypeParser;

// define the type for flex and bison
#define YYSTYPE STypeParser

#endif // __TYPE_PARSER_H__

```

Since in this sample we use the reentrant version of both flex and yacc we are forced to provide parameters for the `yylex` function, when called from `yyparse`.^[6]

```

/*
 * ParserParam.h
 * Definitions of the parameters for the reentrant functions
 * of flex (yylex) and bison (yyparse)
 */

#ifndef __PARSERPARAM_H__
#define __PARSERPARAM_H__

```

```

#ifndef YY_NO_UNISTD_H
#define YY_NO_UNISTD_H 1
#endif // YY_NO_UNISTD_H

#include "TypeParser.h"
#include "Lexer.h"
#include "Expression.h"

/** 
 * @brief structure given as argument to the reentrant 'yyparse'
function.

 */
typedef struct tagSParserParam
{
    yyscan_t scanner;
    SExpression *expression;
} SPParserParam;

// the parameter name (of the reentrant 'yyparse' function)
// data is a pointer to a 'SParserParam' structure
#define YYPARSE_PARAM      data

// the argument for the 'yylex' function
#define YYLEX_PARAM        ((SParserParam*)data)->scanner

#endif // __PARSERPARAM_H__

```

The tokens needed by the bison parser will be generated using flex.

```

%{

/*
 * Lexer.l file
 * To generate the lexical analyzer run: "flex --outfile=Lexer.c
--header-file=Lexer.h Lexer.l"
 */

#include "TypeParser.h"
#include "Parser.h"

%}

%option reentrant noyywrap never-interactive nounistd
%option bison-bridge

LPAREN      "("
RPAREN      ")"

```

```

PLUS          "+"
MULTIPLY     "*"

NUMBER        [0-9] +
WS            [ \r\n\t]*

%%

{WS}           { /* Skip blanks. */ }
{NUMBER}       { sscanf(yytext, "%d", &yyval->value); return
TOKEN_NUMBER; }

{MULTIPLY}     { return TOKEN_MULTIPLY; }
{PLUS}         { return TOKEN_PLUS; }
{LPAREN}       { return TOKEN_LPAREN; }
{RPAREN}       { return TOKEN_RPAREN; }
.

{ }            { }

%%

int yyerror(const char *msg) { fprintf(stderr, "Error:%s\n", msg); return
0; }

```

The bison file providing describing the grammar of the numerical expressions.

```

% {

/*
 * Parser.y file
 * To generate the parser run: "bison --defines=Parser.h Parser.y"
 */

#include "TypeParser.h"
#include "ParserParam.h"

%}

#define api.pure

%left '+' TOKEN_PLUS
%left '*' TOKEN_MULTIPLY

%token TOKEN_LPAREN
%token TOKEN_RPAREN
%token TOKEN_PLUS
%token TOKEN_MULTIPLY

%token <value> TOKEN_NUMBER

```

```
%type <expression> expr

%%

input:
expr { ((SParserParam*)data)->expression = $1; }
;

expr:
expr TOKEN_PLUS expr { $$ = createOperation( ePLUS, $1, $3 ); }
| expr TOKEN_MULTIPLY expr { $$ = createOperation( eMULTIPLY, $1,
$3 ); }
| TOKEN_LPAREN expr TOKEN_RPAREN { $$ = $2; }
| TOKEN_NUMBER { $$ = createNumber($1); }
;

%%
```

The code needed to obtain the syntax tree using the parser generated by bison and the scanner generated by flex is the following.

```
#include "ParserParam.h"
#include "Parser.h"
#include "Lexer.h"

#include <stdio.h>

int yyparse(void *param);

static int initParserParam(SPParserParam* param)
{
    int ret = 0;

    ret = yylex_init(&param->scanner);
    param->expression = NULL;

    return ret;
}

static int destroyParserParam(SPParserParam* param)
{
    return yylex_destroy(param->scanner);
}

SExpression *getAST(const char *expr)
{
    SPParserParam p;
```

```
YY_BUFFER_STATE state;

if ( initParserParam(&p) )
{
    // couldn't initialize
    return NULL;
}

state = yy_scan_string(expr, p.scanner);

if ( yyparse(&p) )
{
    // error parsing
    return NULL;
}

yy_delete_buffer(state, p.scanner);

destroyParserParam(&p);

return p.expression;
}

int evaluate(SExpression *e)
{
    switch(e->type)
    {
        case eVALUE:
            return e->value;
        case eMULTIPLY:
            return evaluate(e->left) * evaluate(e->right);
        case ePLUS:
            return evaluate(e->left) + evaluate(e->right);
        default:
            // shouldn't be here
            return 0;
    }
}

int main(void)
{
    SExpression *e = NULL;
    char test[]=" 4 + 2*10 + 3*( 5 + 1 )";
    int result = 0;

    e = getAST(test);
```

```
    result = evaluate(e);

    printf("Result of '%s' is %d\n", test, result);

    deleteExpression(e);

    return 0;
}
```

Issues

Reentrancy

Normally, Bison generates a parser which is not reentrant. In order to achieve reentrancy the declaration `%define api.pure` must be used. More details on Bison reentrancy can be found in the Bison manual.^[7]

Using bison from other languages

Bison can only generate code for C, C++ and Java.^[8] For using the bison generated parser from other languages a language binding tool such as SWIG can be used.

Where is it used?

Here is a non-comprehensive list of software built using Bison:

- The Ruby Programming Language (YARV);
- The PHP Programming Language (Zend Parser);
- GCC started out using Bison, but switched to a hand-written parser in 2000^[9];
- The Go Programming Language (GC)
- Bash shell uses a yacc grammar for parsing the command input. It is distributed with bison-generated files.

References

- [1] Joel E. Denny (2011-05-15). "-05-012@comp.compilers bison-2.5 released [stable] (news:11)". [news:comp.compilers comp.compilers]. (Web link) (http://groups.google.com/group/comp.compilers/browse_thread/thread/7efb474ecb70bf11/7bf0e228c9534737). Retrieved 2011-06-04.
- [2] http://en.wikipedia.org/wiki/Template%3Alatest_stable_software_release%2Fgnu_bison?action=edit&preload=Template:LSR/syntax
- [3] <http://www.gnu.org/software/bison/>
- [4] Levine, John (August 2009). *flex & bison* (<http://oreilly.com/catalog/9780596155988>). O'Reilly Media. pp. 304. ISBN 978-0-596-15597-1.
- [5] GNU Bison Manual: Conditions for Using Bison (http://www.gnu.org/software/bison/manual/html_node/Conditions.html)
- [6] GNU Bison Manual: C Scanners with Bison Parsers (<http://flex.sourceforge.net/manual/Bison-Bridge.html>)
- [7] GNU Bison Manual: A Pure (Reentrant) Parser (<http://www.gnu.org/software/bison/manual/bison.html#Pure-Decl>)
- [8] GNU Bison Manual: Bison Declaration Summary (http://www.gnu.org/software/bison/manual/html_node/Decl-Summary.html)
- [9] GCC drops YACC (<http://gcc.gnu.org/ml/gcc/2000-10/msg00573.html>)

External links

- Website in the GNU Project (<http://www.gnu.org/software/bison/>)
 - Manual (<http://www.gnu.org/software/bison/manual/>)
- project home at Savannah (<http://savannah.gnu.org/projects/bison/>)
- entry in the Free Software Directory (<http://directory.fsf.org/bison.html>)
- Internals of C parsers generated by GNU Bison (<http://cs.uic.edu/~spopuri/cparser.html>)
- Win32 binaries by GnuWin32 (<http://gnuwin32.sourceforge.net/packages/bison.htm>) (version 2.4.1)

Coco/R

Coco/R

Original author(s)	Hanspeter Mössenböck and others
Platform	Cross-platform
Type	Parser/scanner generator
License	GNU GPL
Website	Coco/R home ^[1]

Coco/R is a compiler generator that takes an L-attributed Extended Backus–Naur Form (EBNF) grammar of a source language and generates a scanner and a parser for that language.

The **scanner** works as a deterministic finite-state machine. It supports Unicode characters in UTF-8 encoding and can be made case-sensitive or case-insensitive. It can also recognize tokens based on their right-hand-side context. In addition to terminal symbols the scanner can also recognize pragmas, which are tokens that are not part of the syntax but can occur anywhere in the input stream (e.g. compiler directives or end-of-line characters).

The **parser** uses recursive descent; LL(1) conflicts can be resolved by either a multi-symbol lookahead or by semantic checks. Thus the class of accepted grammars is LL(k) for an arbitrary k . Fuzzy parsing is supported by so-called ANY symbols that match complementary sets of tokens. Semantic actions are written in the same language as the generated scanner and parser. The parser's error handling can be tuned by specifying synchronization points and "weak symbols" in the grammar. Coco/R checks the grammar for completeness, consistency, non-redundancy as well as for LL(1) conflicts.

There are versions of Coco/R for most modern languages (Java, C#, C++, Pascal, Modula-2, Modula-3, Delphi, VB.NET, Python, Ruby and others). The latest versions from the University of Linz are those for C#, Java and C++. For the Java version, there is an Eclipse plug-in and for C#, a Visual Studio plug-in. There are also sample grammars for Java and C#.

Coco/R was originally developed at the University of Linz and is distributed under the terms of a slightly relaxed GNU General Public License.

References

- Terry, Pat (2005). *Compiling with C# and Java*. Addison Wesley. – A book about using Coco/R for compiler construction.

External links

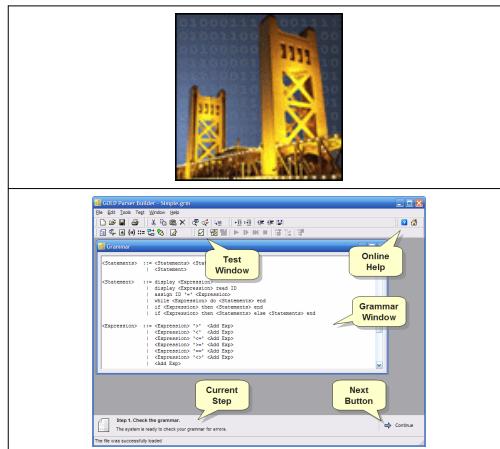
- Coco/R homepage ^[28]
- Coco/R page of Pat Terry ^[2]
- Coco/R user manual ^[3]
- Coco/R tutorial slides (by Hanspeter Mössenböck) ^[4]
- Coco/R Addin for Visual Studio (by Prabir Shrestha) ^[5]

References

- [1] <http://ssw.jku.at/Coco/>
- [2] <http://www.scifac.ru.ac.za/coco/>
- [3] <http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf>
- [4] <http://www.ssw.uni-linz.ac.at/Coco/Tutorial/>
- [5] <http://cocor.codeplex.com/>

GOLD

GOLD Parsing System



Developer(s)	Devin Cook & Multiple Contributors [1]
Stable release	4.0 / 2010-05-28
Operating system	Windows
Type	Parsers - LALR
License	zlib License (free software)
Website	www.devincook.com/goldparser [2]

GOLD is a freeware parsing system that is designed to support multiple programming languages.

Design

The system uses a DFA for lexical analysis and the LALR algorithm for parsing. Both of these algorithms are state machines that use tables to determine actions. GOLD is designed around the principle of logically separating the process of generating the LALR and DFA parse tables from the actual implementation of the parsing algorithms themselves. This allows parsers to be implemented in different programming languages while maintaining the same grammars and development process.

The GOLD system consists of three logical components, the "Builder", the "Engine", and a "Compiled Grammar Table" file definition which functions as an intermediary between the Builder and the Engine.

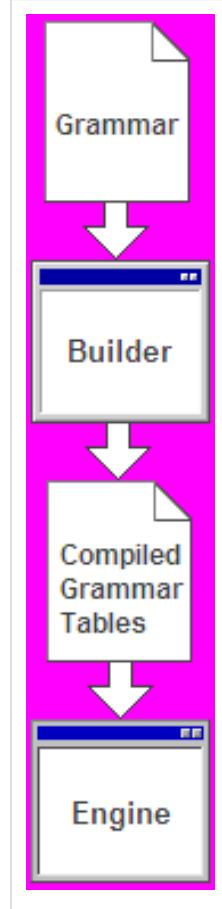
Builder

The Builder is the primary component and main application of the system. The Builder is used to analyze the syntax of a language (specified as a grammar) and construct LALR and DFA tables. During this process, any ambiguities in the grammar will be reported. This is essentially the same task that is performed by compiler-compilers such as YACC and ANTLR.

Once the LALR and DFA parse tables are successfully constructed, the Builder can save this data into a Compiled Grammar Table file. This allows the information to be reopened later by the Builder or used in one of the Engines. Currently, the Builder component is only available for Windows 32-bit operating systems.

Some of the features of the Builder are as follows:

- Freeware license
- State Browsing
- Integrated Testing
- Test multiple files wizard
- Generate Webpages (including hyperlinked syntax charts)
- Generate skeleton programs using templates
- Export grammars to YACC
- Export tables to XML or formatted text



Compiled Grammar Table file

The Compiled Grammar Table file is used to store table information generated by the Builder.

Engines

Unlike the Builder, which only runs on a single platform, the Engine component is written for a specific programming language and/or development platform. The Engine implements the LALR and DFA algorithms. Since different programming languages use different approaches to designing programs, each implementation of the Engine will vary. As a result, an implementation of the Engine written for Visual Basic 6 will differ greatly from one written for ANSI C.

Currently, Engines for GOLD have been implemented for the following programming languages / platforms. New Engines can be implemented using the source code for the existing Engines as the starting point.

- Assembly - Intel x86
- ANSI C
- C#
- D
- Delphi
- Java
- Pascal
- Python
- Visual Basic
- Visual Basic .NET
- Visual C++

Grammars

GOLD grammars are based directly on Backus-Naur Form, regular expressions, and set notation.

The following grammar defines the syntax for a minimal general-purpose programming language called "Simple".

```
"Name"      = 'Simple'
"Author"    = 'Devin Cook'
"Version"   = '2.1'
"About"     = 'This is a very simple grammar designed for use in examples'

"Case Sensitive" = False
"Start Symbol"   = <Statements>

{String Ch 1} = {Printable} - ['']
{String Ch 2} = {Printable} - [""]

Identifier     = {Letter}{AlphaNumeric}*
! String allows either single or double quotes

StringLiteral = ''  {String Ch 1}* ''
                  | """ {String Ch 2}* """

NumberLiteral = {Number}+('.'{Number})?

Comment Start = '/*'
Comment End   = '*/'
Comment Line  = '//''

<Statements> ::= <Statements> <Statement>
                  | <Statement>

<Statement> ::= display <Expression>
                  | display <Expression> read ID
                  | assign ID '=' <Expression>
                  | while <Expression> do <Statements> end
                  | if <Expression> then <Statements> end
                  | if <Expression> then <Statements> else <Statements> end

<Expression> ::= <Expression> '>' <Add Exp>
                  | <Expression> '<' <Add Exp>
                  | <Expression> '<=' <Add Exp>
                  | <Expression> '>=' <Add Exp>
                  | <Expression> '==' <Add Exp>
                  | <Expression> '<>' <Add Exp>
                  | <Add Exp>

<Add Exp>      ::= <Add Exp> '+' <Mult Exp>
```

```

| <Add Exp> '-' <Mult Exp>
| <Add Exp> '&' <Mult Exp>
| <Mult Exp>

<Mult Exp> ::= <Mult Exp> '*' <Negate Exp>
| <Mult Exp> '/' <Negate Exp>
| <Negate Exp>

<Negate Exp> ::= '-' <Value>
| <Value>

<Value> ::= Identifier
| StringLiteral
| NumberLiteral
| '(' <Expression> ')'

```

Development overview

Design the grammar

The first step consists of writing and testing a grammar for the language being parsed. The grammar can be written using any text editor - such as Notepad or the editor that is built into the Builder. At this stage, no coding is required.

Construct the tables

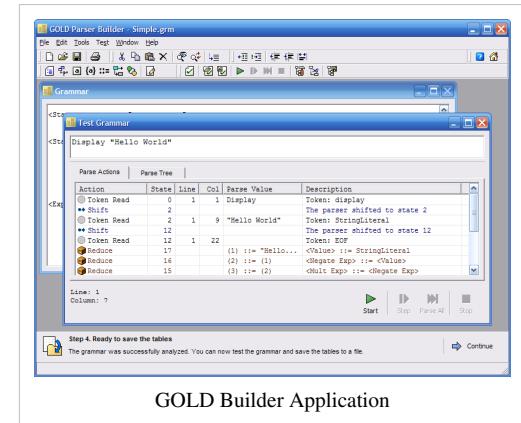
Once the grammar is complete, it is analyzed by the Builder, the LALR and DFA parse tables are constructed, and any ambiguities or problems with the grammar are reported. Afterwards, the tables are saved to a Compiled Grammar Table file to be used later by a parsing engine. At this point, the GOLD Parser Builder is no longer needed.

Select a parsing Engine

In the final stage, the tables are read by an Engine. At this point, the development process is dependent on the selected implementation language.

External links

- GOLD Parsing System Homepage ^[3]
 - List of Contributors ^[1]
- GOLD Yahoo Group ^[4]



References

- [1] <http://www.devincook.com/goldparser/contributors>
- [2] <http://www.devincook.com/goldparser>
- [3] <http://www.devincook.com/goldparser/>
- [4] <http://tech.groups.yahoo.com/group/goldparser/>

JavaCC

JavaCC

Stable release	5.0 / August 31, 2009
Platform	Java Virtual Machine
Type	parser/scanner generator
License	BSD
Website	http://javacc.dev.java.net/

JavaCC (Java Compiler Compiler) is an open source parser generator and lexical analyzer generator for the Java programming language. JavaCC is similar to yacc in that it generates a parser from a formal grammar written in EBNF notation, except the output is Java source code. Unlike yacc, however, JavaCC generates top-down parsers, which limits it to the LL(k) class of grammars (in particular, left recursion cannot be used). JavaCC also generates lexical analyzers in a fashion similar to lex. The tree builder that accompanies it, JJTree, constructs its trees from the bottom up.

JavaCC is licensed under a BSD license.

History

In 1996, Sun Microsystems released a parser generator called *Jack*. The developers responsible for *Jack* created their own company called Metamata and changed the *Jack* name to JavaCC. Metamata eventually became part of WebGain. After WebGain shut down its operations, JavaCC was moved to its current home.

External links

- Official JavaCC website ^[1] - New site (as of January 2011) at java.net/projects/javacc
- Old Official JavaCC web site ^[46] - Apparently most of the site was (re)moved...?
- A working snapshot of the old official website ^[2] - Snapshot archived in 2008 by Internet Archives. Includes more useful content than the current state of the new site.
- JavaCC Tutorial ^[3]
- JavaCC FAQ ^[4]
- A JavaCC book - Generating Parsers with JavaCC ^[5]

References

- [1] <http://java.net/projects/javacc>
- [2] <http://replay.waybackmachine.org/20080924103934/https://javacc.dev.java.net/>
- [3] <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/>
- [4] <http://www.engr.mun.ca/~theo/JavaCC-FAQ/>
- [5] <http://generatingparserswithjavacc.com/>

JetPAG

JetPAG

Developer(s)	Tareq H. Sharafy
Stable release	0.6.1 / February 7, 2007
Preview release	0.6.3 / 2007
Written in	C++
Operating system	Platform-independent
Type	Parser generator
License	GNU General Public License
Website	JetPAG Homepage ^[49]

JetPAG (**J**et **P**arser **A**uto-**G**enerator) is an open source LL(k) parser and lexical analyzer generator, licensed under the GNU General Public License. It is a personal work of Tareq H. Sharafy, and is currently at final beta stages of development.

History

Tareq started JetPAG as a small program written for practice purposes only. Soon when it started expanding many goals were added rapidly, and it was obvious that JetPAG is worthy being a complete project. Real development of JetPAG started in late 2005, targeting a complete framework for a powerful recursive descent lexical analyzer and parser generator with emphasis on ease of use, code readability and high performance of generated code. After a long period of in-house development and testing, the first development package of JetPAG was released through SourceForge in 18 November 2006. Development of JetPAG is current at beta stage, current version is 0.6.1. The development was delayed from mid-2007 until early 2009 but resumed after.

Overview

Jetpag incorporates several modules: the front end, the analyzers and the code generators.

The front end accepts the grammar metalanguages as an input.

The analyzers mainly perform two operations through tree traversal. The first is calculating strong lookahead sets for the elements in the grammar and the second is constructing lookahead paths from the lookahead sets. Lookahead paths group, factorize and perform many enhancements and optimizations to lookahead sets using special analysis. From lookahead paths lookahead sets are transformed to a nested tree form, gaining a great overall efficiency and improvement in most cases.

Code generators generate source code for recognizers compatible with the input grammars based on them along with information collected from the analyzers. Currently JetPAG generates source code in C++ only.

The nature of JetPAG's metalanguage and framework make it easy and simple to integrate generated recognizers into larger applications. JetPAG also includes some facilities in the provided framework to aid developers with small utilities and save development time from many minimal language recognition tasks.

JetPAG grammars

JetPAG's grammars are written in a meta language based on the EBNF form and regular expressions, with extensive additions and tweaks. The meta language of JetPAG grammars was designed to be maximally flexible handle both simple grammars and large, complicated ones easily. Parsers and lexical analyzers are similarly defined and generated for simplicity and ease of use. This is a simple example of a grammar for a basic calculator:

```
grammar Calc;

parser CalcP:

expression:
multiplicative
( '+' multiplicative
| '-' multiplicative
)*
;

multiplicative:
factor
( '*' factor
| '/' factor
)*
;

factor:
INT
| '(' expression ')'
;

scanner CalcS:

INT:   '0'-'9'+;
PLUS:  '+';
MINUS: '-';
STAR:  '*';
SLASH: '/';
LP:    '(';
RP:    ')';
```

External links

- JetPAG Homepage^[49]
- JetPAG at SourceForge^[1]

References

- [1] <http://sourceforge.net/projects/jetpag/>
-

Lemon Parser Generator

Lemon Parser Generator

Developer(s)	D. Richard Hipp
Written in	C
Operating system	Cross-platform
Type	Parser generator
License	Public domain
Website	http://www.hwaci.com/sw/lemon/ ^[1]

Lemon is a parser generator, maintained as part of the SQLite project, that generates an LALR parser in the C programming language from an input context-free grammar. The generator is quite simple, implemented in a single C source file with another file used as a template for output. Lexical analysis is performed externally.

Lemon is similar to bison and yacc; however it is not compatible with these programs. The grammar input format is different to help prevent common coding errors. Other distinctive features include an output parser that is reentrant and thread-safe, and the concept of "non-terminal destructors" that try to make it easier to create a parser that does not leak memory.

SQLite uses Lemon with a hand-coded tokenizer to parse SQL strings.

In 2008 a Lemon-generated parser was suggested to replace the bison-generated parser used for the PHP programming language; as of 2010 this project is listed as "in the works".^[2]

Notes

[1] <http://www.hwaci.com/sw/lemon/>

[2] Kneuss, Etienne (2008-03-25). "Request for Comments: Replace Bison based parser with Lemon" (<http://wiki.php.net/rfc/lemon>). *PHP Wiki*. . Retrieved 2010-05-08.

References

- "The LEMON Parser Generator" (<http://www.hwaci.com/sw/lemon>). Retrieved 2008-12-24.
- "Architecture of SQLite" (<http://www.sqlite.org/arch.html>). 2008-11-01. Retrieved 2008-12-24.

External links

- The Lemon Parser Generator (<http://www.hwaci.com/sw/lemon>)
- Calculator with Lemon and Lex in C++ Example (<http://freshmeat.net/articles/view/1270/>)
- Understanding Lemon generated Parser (<http://www.gnudeveloper.com/groups/lemon-parser/understanding-lemon-generated-parser.html>)

ROSE compiler framework

ROSE

Developer(s)	Lawrence Livermore National Laboratory
Written in	C++
Operating system	Linux
Type	Compiler
License	BSD licenses
Website	rosecompiler.org ^[1]

The **ROSE** compiler framework, developed at Lawrence Livermore National Laboratory (LLNL), is an open source compiler infrastructure to generate source-to-source analyzers and translators for multiple source languages including C, C++, and Fortran. It also supports OpenMP, UPC and certain binary files. Unlike most other research compilers, ROSE is aimed to enable non-experts to leverage compiler technologies to build their own custom software analyzers and optimizers.

The infrastructure

ROSE consists of multiple front-ends, a midend operating on its internal intermediate representation (IR), and backends regenerating (unparse) source code from IR. Optionally, vendor compilers can be used to compile the unparsed source code into final executables.

ROSE uses the Edison Design Group's C++ front-end^[2] to parse C and C++ applications. Fortran support, including F2003 and earlier F77/90/95 versions, is based on the Open Fortran Parser (OFP)^[3] developed at Los Alamos National Laboratory.

The ROSE IR consists of an abstract syntax tree, symbol tables, control flow graph, etc. It is an object-oriented IR with several levels of interfaces for quickly building source-to-source translators. All information from the input source code is carefully preserved in the ROSE IR, including C preprocessor control structure, source comments, source position information, and C++ template information (e.g., template arguments).

ROSE is released under a BSD-style license. It targets Linux and Mac OS X on both IA-32 and x86-64 platforms. Its EDG parts are proprietary and distributed in binary form. Source files of the EDG parts can be obtained if users have a commercial or research license from EDG.

Award

The ROSE compiler infrastructure has received one of the 2009 R&D 100 Awards. The R&D 100 Awards are presented annually by *R&D Magazine* to recognize the 100 most significant proven research and development advances introduced over the past year. An independent expert panel selects the winners.

External links

- Official website^[1]
- Development site^[4]

Other Source-to-Source compilers

- DMS Software Reengineering Toolkit
- Stratego/XT
- TXL

References

- [1] <http://www.rosecompiler.org/>
- [2] http://www.edg.com/index.php?location=c_frontend
- [3] <http://fortran-parser.sourceforge.net/>
- [4] <https://outreach.scidac.gov/projects/rose/>

SableCC

SableCC

Stable release	3.2
Preview release	4-beta.2
Written in	Java
Platform	Java Virtual Machine
Type	Parser/scanner generator
License	GNU Lesser General Public License
Website	http://www.sablecc.org/

SableCC is an open source compiler generator (or interpreter generator) in Java. Stable version is licensed under the GNU Lesser General Public License (LGPL). Rewritten version 4 is licensed under Apache License 2.0.

SableCC includes the following features:

- Deterministic finite automaton (DFA)-based lexers with full Unicode support and lexical states.
- Extended Backus-Naur Form grammar syntax. (Supports the *, ? and + operators).
- LALR(1) based parsers.
- Automatic generation of strictly-typed abstract syntax trees.
- Automatic generation of tree-walker classes.

External links

- SableCC website ^[1]

References

[1] <http://www.sablecc.org/>

Scannerless Boolean Parser

SBP

Developer(s)	Adam Megacz
Platform	Java Virtual Machine
Type	Parser generator
License	BSD license
Website	[140]

The **Scannerless Boolean Parser** is a free software scannerless GLR parser generator for boolean grammars. It was implemented in the Java programming language and generates Java source code. SBP also integrates with Haskell via LambdaVM.

External links

- SBP: the Scannerless Boolean Parser [140]
- SBP LDTA'06 article [1]
- W_IX - wiki markup parser in SBP Haskell [2]

References

- [1] <http://www.megacz.com/research/megacz.adam-sbp.a.scannerless.boolean.parser.pdf>
- [2] <http://www.megacz.com/software/wix/>

Spirit Parser Framework

The **Spirit Parser Framework** is an object oriented recursive descent parser generator framework implemented using template metaprogramming techniques. Expression templates allow users to approximate the syntax of Extended Backus Naur Form (EBNF) completely in C++. Parser objects are composed through operator overloading and the result is a backtracking LL(∞) parser that is capable of parsing rather ambiguous grammars.

Spirit can be used for both lexing and parsing, together or separately.

This framework is part of the Boost libraries.

Operators

Because of limitations of the C++ language, the syntax of Spirit has been designed around the operator precedences of C++, while bearing resemblance to both EBNF and regular expressions.

syntax	explanation
x >> y	Match x followed by y.
*x	Match x repeated zero or more times. (This is representing the Kleene star; C++ lacks a unary postfix operator *)
x y	Match x. If x does not match, try to match y.
+x	Match x repeated one or more times.
-x	Match x zero or one time.
x & y	Match x and y.
x - y	Match x but not y.
x ^ y	Match x or y or both in any order.
x y	Match x or y or x followed by y.
x [function_expression]	Execute the function/functor returned by function_expression, if x matched.
(x)	Match x (can be used for priority grouping)
x % y	Match one or more repetitions of x, separated by occurrences of y.
~x	Match anything but x (only with character classes such as ch_p or alnum_p)

Example

```
#include <boost/spirit/include/classic_core.hpp>
#include <boost/spirit/include/classic_increment_actor.hpp>
#include <string>
#include <iostream>

using namespace std;
using namespace BOOST_SPIRIT_CLASSIC_NS;

int main()
{
    string input;

    cout << "Input a line." << endl;
```

```
getline(cin, input);

cout << "Got '" << input << "'." << endl;

unsigned count = 0;

/*
Next line parses the input (input.c_str()),
using a parser constructed with the following semantics
(indentation matches source for clarity):

Zero or more occurrences of (
literal string "cat" ( when matched, increment the counter
"count" )
or any character (to move on finding the next occurrence of
"cat")
)
parse(input.c_str(),
*( str_p("cat") [ increment_a(count) ]
| anychar_p
)) ;
*/

The parser is constructed by the compiler using operator
overloading and template matching, so the actual work is
done within spirit::parse(), and the expression starting
with * only initializes the rule object that the parse
function uses.
*/

// last, show results.
cout << "The input had " << count << " occurrences of 'cat'" << endl;
return 0;
}
```

Of course, there are better algorithms suited for string searching, but this example gives an idea how to construct rules and attach actions to them.

External links

- Spirit parser framework SourceForge page ^[1]
- Documentation in the Boost project ^[2]
- Article on Spirit by designer Joel de Guzman in Dr. Dobb's Journal ^[3]

References

- [1] <http://spirit.sourceforge.net>
 [2] <http://www.boost.org/libs/spirit/index.html>
 [3] http://www.ddj.com/article/printableArticle.jhtml?articleID=184401692&dept_url=/cpp/

S/SL programming language

The **Syntax/Semantic Language (S/SL)** is an executable high level specification language for recursive descent parsers, semantic analyzers and code generators developed by James Cordy, Ric Holt and David Wortman at the University of Toronto in 1980.^[1]

S/SL is a small programming language that supports cheap recursion and defines input, output, and error token names (& values), semantic mechanisms (class interfaces whose methods are really escapes to routines in a host programming language but allow good abstraction in the pseudo-code) and a pseudo-code program that defines the syntax of the input language by the token stream the program accepts. Alternation, control flow and one-symbol look-ahead constructs are part of the language.

The S/SL processor compiles this pseudo-code into a table (byte-codes) that is interpreted by the S/SL table-walker (interpreter). The pseudo-code language processes the input language in LL(1) recursive descent style but extensions allow it to process any LR(k) language relatively easily.^[2] S/SL is designed to provide excellent syntax error recovery and repair. It is more powerful and transparent than Yacc but can be slower.

S/SL's "semantic mechanisms" extend its capabilities to all phases of compilation, and it has been used to implement all phases of compilation, including scanners, parsers, semantic analyzers, code generators and virtual machine interpreters in multi-pass language processors.^[3]

S/SL has been used to implement production commercial compilers for languages such as PL/I, Euclid, Turing, Ada, and COBOL, as well as interpreters, command processors, and domain specific languages of many kinds. It is the primary technology used in IBM's ILE/400 COBOL compiler,^[4] and the ZMailer mail transfer agent uses S/SL^[5] for defining both its mail router processing language and its RFC 822 email address validation.

References

- [1] J. R. Cordy, R. C. Holt and D. B. Wortman, "S/SL: Syntax/Semantic Language - Introduction and Specification, Technical Report CSRG-118, Computer Systems Research Group, University of Toronto, Sept. 1980
- [2] D.T. Barnard and J.R. Cordy, "SL Parses the LR Languages", Computer Languages 13,2 (April 1988), pp. 65-74 [http://dx.doi.org/10.1016/0096-0551\(88\)90010-0](http://dx.doi.org/10.1016/0096-0551(88)90010-0)
- [3] Richard C. Holt, James R. Cordy and David B. Wortman, "An Introduction to S/SL: Syntax/Semantic Language", ACM Transactions on Programming Languages and Systems 4,2 (April 1982) <http://doi.acm.org/10.1145/357162.357164>
- [4] Ian H. Carmichael and Stephen Perelgut. S/SL revisited. Proc. CASCON'95, Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Canada, November 1995 <http://portal.acm.org/citation.cfm?id=781915.781926>
- [5] ZMailer the Manual, <http://www.zmailer.org/zman/zmanual.shtml>

SYNTAX

SYNTAX

Developer(s)	INRIA
Type	Generator
License	CeCILL
Website	syntax.gforge.inria.fr ^[1]

In computer science, **SYNTAX** is a system used to generate lexical and syntactic analyzers (parsers) (both deterministic and non-deterministic) for all kind of context-free grammars (CFG) as well as some classes of contextual grammars. It is developed at INRIA (France) for several decades, mostly by Pierre Boullier, but has become free software since 2007 only. SYNTAX is distributed under the CeCILL license.

Context-free parsing

SYNTAX handles most classes of deterministic (unambiguous) grammars (LR, LALR, RLR) as well as general context-free grammars. The deterministic version has been used in operational contexts (e.g., Ada^[2]), and is currently used both in the domain of compilation^[3]. The non-deterministic features include an Earley parser generator used for natural language processing^[4]. Parsers generated by SYNTAX include powerful error recovery mechanisms, and allow the execution of semantic actions and attribute evaluation on the abstract tree or on the shared parse forest.

Contextual parsing

The current version of SYNTAX (version 6.0 beta) includes also parser generators for other formalisms, used for natural language processing as well as bio-informatics. These formalisms are context-sensitive formalisms (TAG, RCG) or formalisms that rely on context-free grammars and are extended thanks to attribute evaluation, in particular for natural language processing (LFG).

Notes

[1] <http://syntax.gforge.inria.fr/>

[2] The first tool-translator for the ADA language has been developed with SYNTAX by Pierre Boullier and others, as recalled in this page on the history of ADA (<http://archive.adaic.com/standards/83lrm/html/lrm-FORE.html>). See also Pierre Boullier and Knut Ripken. Building an Ada compiler following meta-compilation methods. In Séminaires Langages et Traducteurs 1978-1981, pages 99-140. INRIA, Rocquencourt, France, 1981.

[3] E.g., by the VASY (<http://www.inrialpes.fr/vasy>) team at INRIA, in particular for the development of CADP (<http://www.inrialpes.fr/vasy/cadp/>) and Traian (<http://www.inrialpes.fr/vasy/traijan/>).

[4] E.g., in the SxLFG parser, whose first version is described in this paper (<http://atoll.inria.fr/~sagot/pub/IWPT05.pdf>).

External links

- (French) SYNTAX web site (<http://syntax.gforge.inria.fr/>)
- Paper on the construction of compilers using SYNTAX and TRAIAN (Compiler Construction'02 Conference (<http://www.inrialpes.fr/vasy/Publications/Garavel-Lang-Mateescu-02.html>)

Syntax Definition Formalism

The **Syntax Definition Formalism (SDF)** for short) is a metasyntax used to define context-free grammars: that is, a formal way to describe formal languages. It can express the entire range of context-free grammars. Its current version is SDF2. A parser and parser generator for SDF specifications are provided as part of the free ASF+SDF Meta Environment. These operate using the SGLR (Scannerless GLR parser). An SDF parser outputs parse trees or, in the case of ambiguities, parse forests.

Overview

Features of SDF:

- Supports the entire range of context-free languages
- Allows modular syntax definitions (grammars can import subgrammars) which enables reuse
- Supports annotations

Examples

The following example defines a simple Boolean expression syntax:

```
module basic/Booleans

exports
    sorts Boolean
    context-free start-symbols Boolean

context-free syntax
    "true"                  -> Boolean
    "false"                 -> Boolean
    lhs:Boolean " | " rhs:Boolean -> Boolean {left}
    lhs:Boolean " & " rhs:Boolean -> Boolean {left}
    "not" "(" Boolean ")"     -> Boolean
    "(" Boolean ")"          -> Boolean

context-free priorities
    Boolean " & " Boolean -> Boolean >
    Boolean " | " Boolean -> Boolean
```

Program analysis and transformation systems using SDF

- ASF+SDF Meta Environment provides SDF
- RascalMPL
- Spoofax/IMP [1]
- Stratego/XT
- Strafunski

Further reading

- A Quick Introduction to SDF, Visser, J. & Scheerder, J. (2000) CWI^[2]

External links

- Grammar Deployment Kit^[3]
- SdfMetz^[4] computes metrics for SDF grammars
- Download SDF from the ASF+SDF Meta Environment homepage^[5]

References

- [1] <http://strategoxt.org/Spoofax>
- [2] <ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf>
- [3] <http://gdk.sourceforge.net/>
- [4] <http://wiki.di.uminho.pt/twiki/bin/view/Research/PURE/SdfMetz>
- [5] <http://www.meta-environment.org/>

TREE-META

TREE-META

Original author(s)	D. V. Shorre?, Donald Andrews, and Jeff Rulifson
Initial release	1968?
Development status	unknown

The **TREE-META** (aka Tree Meta and TREEMETA) Translator Writing System is a Compiler-compiler system for context-free languages originally developed in the 1960s. Parsing statements of the metalanguage resemble Backus-Naur Form with embedded tree-building directives. Output producing rules include extensive tree-scanning and code-generation constructs.

History

TREE-META was instrumental in the development of the On-Line System and was ported to many systems including the Univac 1108, GE 645, SDS-940, ICL 1906A, PERQ, and UCSD p-System.^[1]

TREE-META was the last of a line of metacompilers, starting with META II, right before subsequent versions were designated classified technology by the U.S. military and government agencies.

Example

This is a complete example of a TREE-META program extracted (and untested) from the more complete (declarations, conditionals, and blocks) example in Appendix 6 of the ICL 1900 TREE-META manual.^[2] That document also has a definition of TREE-META in TREE-META in Appendix 3. This program is not just a recognizer, but also outputs the assembly language for the input. It demonstrates one of the key features of TREE-META which is tree pattern matching. It is used on both the LHS (GET and VAL for example) and the RHS (ADD and SUB). This example doesn't show the other key feature (and what distinguishes TREE-META from the other META II based languages) which is tree transforming rules.

```
.META PROG
$ THIS RULE DEFINES SYNTAX OF COMPLETE PROGRAM $

PROG   =  STMT * ;
STMT   = .ID ':=' AEXP :STORE[2] ;
AEXP   = FACTOR $ ( '+' FACTOR :ADD[2] / '-' FACTOR :SUB[2] );
FACTOR = '-' PRIME :MINUSS[1] / PRIME ;
PRIME  = .ID / .NUM / '(' AEXP ')' ?3? ;

$ OUTPUT RULES $

STORE[-,-] => GET[*2] 'STORE' *1 % ;

GET[.ID] => 'LOAD' *1 %
[.NUM] => 'LOADI' *1 %
[MINUSS[.NUM]] => 'LOADN' *1:*1 %
[-]    => *1 ;
```

```

ADD [-,-] => SIMP[*2] GET[*1] 'ADD' VAL[*2] % /
    SIMP[*1] GET[*2] 'ADD' VAL[*1] % /
    GET[*1] 'STORE T+' < OUT[A] ; A<-A+1 >%
    GET[*2] 'ADD T+' < A<-A-1 ; OUT[A] > % ;

SUB [-,-] => SIMP[*2] GET[*1] 'SUB' VAL[*2] % /
    SIMP[*1] GET[*2] 'NEGATE' % 'ADD' VAL[*1] % /
    GET[*2] 'STORE T+' < OUT[A] ; A<-A+1 > %
    GET[*1] 'SUB T+' < A<-A-1 ; OUT[A] > % ;

SIMP [.ID] => .EMPTY
[.NUM] => .EMPTY
[MINUSS [.NUM]] => .EMPTY;

VAL [.ID] => ' ' *1
[.NUM] => 'I ' *1
[MINUSS [.NUM]] => 'N ' *1:*1 ;

MINUSS [-] => GET[*1] 'NEGATE' %;

.END

```

References

- [3] C. Stephen Carr, David A. Luther, Sherian Erdmann, 'The TREE-META Compiler-Compiler System: A Meta Compiler System for the Univac 1108 and General Electric 645', University of Utah Technical Report RADC-TR-69-83.
- [4], also [5] 1968 Tech Report by Englebart, English, and Rulifson on Tree Meta's use in what they called Special-Purpose Languages (SPL's), which we now call Domain Specific Languages (DSL's), in the NLS.
- Andrews, Donald I. J. F. Rulifson (1967). Tree Meta (Working Draft): A Meta Compiler for the SDS 940 . , Stanford Research Institute, Menlo Park, CA. Engelbart Collection, Stanford University Archive, M 638, Box 16, Folder 3.
- ANDREWS, LEHTMAN, and WHP. "Tree Meta -- a metacompiler for the Augmentation Research Center." Preliminary draft, 25 March 1971.
- [6] Alan C. Kay "The Reactive Engine" Ph.D. thesis 1969 University of Utah. Notes that Henri Gouraud did the FLEX compiler in TREE-META on the SRI (Engelbart) SDS-940.
- [7] Atlas Computer Laboratory quarterly report (21 November 1975), F R A Hopgood documents work using TREE-META to create a compiler generating FR80 assembler output.
- [8] Atlas Computer Laboratory quarterly report (12 October 1973), C J Pavelin documents (section 4.10) TREE-META being ported to the 1906A.
- TREE-META: a meta-compiler for the Interdata Model 4 by W M Newman. Queen Mary College, London. November 1972.

[1] Bowles, K.L., 1978. A (nearly) machine independent software system for micro and mini computers. SIGMINI News!, 4(1), 3-7.
DOI:10.1145/1041256.1041257 (<http://doi.acm.org/10.1145/1041256.1041257>)

[2] Hopgood, F R A 1974, "TREE-META Manual", Atlas Computer Laboratory.

[3] <http://www.dtic.mil/srch/doc?collection=t2&id=AD0855122>

[4] <http://www.dtic.mil/srch/doc?collection=t2&id=AD0843577>

- [5] <http://www.stormingmedia.us/77/7753/0775348.html>
- [6] <http://www.mprove.de/diplom/gui/kay69.html#IV>
- [7] http://www.chilton-computing.org.uk/acl/literature/progress/basic_progress/q3_75.htm
- [8] http://www.chilton-computing.org.uk/acl/literature/progress/basic_progress/q3_73.htm

External links

- Manual for ICL 1900 version of TREE-META by F R A Hopgood. (<http://www.chilton-computing.org.uk/acl/literature/manuals/tree-meta/contents.htm>)
- Home page for collecting information about TREE-META (<http://www.ifcx.org/wiki/TREEMETA.html>)
- TREE META Draft Document December, 1967 at bitsavers.org (http://bitsavers.org/pdf/sri/arc/rulifson/Tree_Meta_A_Meta_Compiler_System_For_The_SDS_940_Dec67.pdf)
- TREE META Release Document April, 1968 at bitsavers.org (http://bitsavers.org/pdf/sri/arc/rulifson/A_Tree_Meta_For_The_XDS_940_Appendix_D_Apr68.pdf)

Frameworks supporting the polyhedral model

Use of the polyhedral model within a compiler requires software to represent the objects of this framework (sets of integer-valued points in regions of various spaces) and perform operations upon them (e.g., testing whether the set is empty).

For more detail about the objects and operations in this model, and an example relating the model to the programs being compiled, see the polyhedral model page.

There are many **frameworks supporting the polyhedral model**. Some of these frameworks use one or more libraries for performing polyhedral operations. Others, notably Omega, combine everything in a single package. Some commonly used libraries are the Omega Library^[1] (and a more recent fork^[2]), piplib^{[3][4]}, PolyLib^{[5][6]}, PPL^[7], isl^[8], the cloog polyhedral code generator^{[3][9]}, and the barvinok library for counting integer solutions^[10]. Of these libraries, PolyLib and PPL focus mostly on rational values, while the other libraries focus on integer values. The polyhedral framework of gcc is called Graphite^[11]. Polly^[12] provides polyhedral optimizations for LLVM.

Common Strengths

Polyhedral frameworks are designed to support compilers techniques for analysis and transformation of codes with nested loops, producing exact results for loop nests with affine loop bounds and subscripts ("Static Control Parts" of programs). They can be used to represent and reason about *executions* (iterations) of statements, rather than treating a statement as a single object representing properties of all executions of that statement. Polyhedral frameworks typically also allow the use of symbolic expressions.

Polyhedral frameworks can be used for dependence analysis for arrays, including both traditional alias analysis and more advanced techniques such as the analysis of data flow in arrays or identification of conditional dependencies. They can also be used to represent code transformation, and provide features to generate the transformed code in a high-level language. The transformation and generation systems can typically handle imperfectly nested loops.

An example to contrast polyhedral frameworks with prior work

To compare the constraint-based polyhedral model to prior approaches such as individual loop transformations and the unimodular approach, consider the question of whether we can parallelize (execute simultaneously) the iterations of following contrived but simple loop:

```
for i := 0 to N do
    A(i) := (A(i) + A(N-i))/2
```

Approaches that cannot represent symbolic terms (such as the loop-invariant quantity N in the loop bound and subscript) cannot reason about dependencies in this loop. They will either conservatively refuse to run it in parallel, or in some cases speculatively run it completely in parallel, determine that this was invalid, and re-execute it sequentially.

Approaches that handle symbolic terms but represent dependencies via direction vectors or distance vectors will determine that the i loop carries a dependence (of unknown distance), since for example when N=10 iteration 0 of the loop writes an array element (A(0)) that will be read in iteration 10 (as A(10-10)) and reads an array element (A(10-0)) that will later be overwritten in iteration 10 (as A(10)). If all we know is that the i loop carries a dependence, we once again cannot safely run it in parallel.

In reality, there are only dependencies from the first N/2 iterations into the last N/2, so we can execute this loop as a sequence of two fully parallel loops (from 0...N/2 and from N/2+1...N). The characterization of this dependence, the analysis of parallelism, and the transformation of the code can be done in terms of the instance-wise information provided by any polyhedral framework.

Instance-wise analysis and transformation allows the polyhedral model to unify additional transformations (such as index set splitting, loop peeling, tiling, loop fusion or fission, and transformation of imperfectly nested loops) with those already unified by the unimodular framework (such as loop interchange, skewing, and reversal of perfectly nested loops). It has also stimulated the development of new transformations, such as Pugh and Rosser's iteration-space slicing (an instance-wise version of program slicing; note that the code was never released with the Omega Library).

A more interesting example

Authors of polyhedral frameworks have explored the simple 1-dimensional finite difference heat equation stencil calculation expressed by the following pseudocode:

```
for t := 0 to T do
    for i := 1 to N-1 do
        new(i) := (A(i-1) + A(i) + A(i) + A(i+1)) * .25 // explicit forward-difference with R = 0.25
    end
    for i := 1 to N-1 do
        A(i) := new(i)
    end
end
```

This code confounds many of the transformation systems of the 20th century, due to the need to optimize an imperfect loop nest. Polyhedral frameworks can analyze the flow of information among different executions of statements in the loop nest, and transform this code to simultaneously exploit scalable parallelism and scalable locality.

A re-cap here, of the two approaches on this example, might be nice, but for now see the individual papers of Wonnacott^[13] [14], and Sadayappan et al.^[15], as well as others who have studied this code using different frameworks, such as Song and Li^[16].

Differences in Presentation or Vocabulary

Comparison of works using different frameworks is complicated by both technical differences (discussed later) and differences in vocabulary and presentation. Examples are provided below to aid in translation:

Classification of Dependences

Polyhedral Frameworks support dependence analysis in a variety of ways, helping to capture the impact of symbolic terms, identify conditional dependences, and separating out the effects of memory aliasing. The effects of memory aliasing, in particular, have been described in two ways: many authors distinguish between "true" data dependences (corresponding to actual flow of information) from false dependences arising from memory aliasing or limited precision of dependence analysis.

The Omega Project publications use specific terms to identify specific effects on analysis. They maintain the traditional distinction of flow-, output-, and anti-dependences, based on the types of array access (write to read, write to write, or read to write, respectively). *Dependences* can independently be classified as memory-based or value-based --- the former corresponds to memory aliasing, and the latter does not include dependences interrupted by intervening writes. A *dependence test* may produce information that is exact or approximate, depending on the nature of the program being analyzed and the algorithms used in the test. Finally, the results of dependence analysis will be reported in a *dependence abstraction* that provides a certain degree of precision.

For example, the "dependence relations" produced by the Omega Test, and the "quasts" produced by the algorithms of Feautrier or Maydan and Lam, contain precise information (though in different forms) about the loop iterations involved in a dependence. The results of either test can be converted into the more traditional "dependence vector" form, but since this abstraction provides less precision, much of the information about the dependence will be lost. Both techniques produce exact information for programs with affine control and subscript expressions, and must approximate for many programs outside this domain (i.e., in the presence of non-affine subscripts such as index arrays). The original work of Feautrier focused on describing *true* dependences, which would be referred to as *exact value-based flow dependences* by the Omega Project. The Omega Project also described the use of their algorithms for value-based output- and anti-dependences, though Feautrier's quasts could presumably be easily adapted to this as well.

Visualization of Transformations and Tiling

There are many ways to produce a visual depiction of the process of transforming and tiling an iteration space. Some authors depict transformations by changing the location of points on the page, essentially aligning the picture with the coordinate axes of the transformed space; in such diagrams, tiles appear as axis-aligned rectangles/rectangular solids containing iterations. Examples of this approach can be found in the publications and transformation-visualization software of Michelle Mills Strout^[17].

Other authors depict different transformations as different wavefronts of execution that move across the points of the original coordinate system at different angles. In such diagrams, tiles appear as parallelograms/parallelepipeds. Examples of this approach can be found in the *time-skewing* publications of David G. Wonnacott^[18].

Differences in Approach or Implementation Status

Some of the libraries have seen more extensive development than the Omega Library in the early 2000s, and in many places has much more sophisticated algorithms. In particular, users have reported good results with the Cloog code generator (both in terms of the code generated, and in terms of ability to control trade-offs when generating code), and with the algorithms for counting integer solutions (Alexander Barvinok^[19]'s work requires a vertex description of the polytope, which is not supported in the Omega Library).

There are several other points on which the frameworks differ, specifically:

Precision and speed

Integer programming is NP-complete, and Maydan showed that the problem of checking array aliasing in nested loops with affine bounds and subscripts is equivalent to integer programming; other operations, such as array dataflow analysis, are even more complex (the algorithms of the Omega Library handle the full language of Presburger Arithmetic, which is $O(2^2 \cdot 2^n)$). Thus, it is clearly unrealistic to expect exact fast results for arbitrary problems of array aliasing or array data flow, even over the affine domain. Fortunately, many problems fall into a subset of this domain where general algorithms can produce an exact answer in polynomial time [20], [21].

Outside of this domain, the Omega Library, piplib and isl emphasize the production of an exact result (except in the cases of certain uses of uninterpreted function symbols in Omega), despite the high complexity. In some cases, such as variable elimination ("projection"), PolyLib and PPL primarily use algorithms for the rational domain, and thus produce an approximation of the result for integer variables. It may be the case that this reduces the common experience with the Omega Library in which a minor change to one coefficient can cause a dramatic shift in the response of the library's algorithms.

Polylib has some operations to produce exact results for Z-polyhedra (integer points bounded by polyhedra), but at the time of this writing, significant bugs have been reported [22]. Note that bugs also exist in the Omega Library, including reliance on hardware-supplied integer types and cases of the full Presburger Arithmetic algorithms that were not implemented in the library. Users who need exact results for integer variables may need to be wary with either library.

Barvinok's techniques for counting integer solutions require a description of the vertices (and bounding rays) of the polyhedron, but produce an exact answer in a way that can be far more efficient than the techniques described by Pugh. Barvinok's algorithm is always polynomial in the input size, for fixed dimension of the polytope and fixed degree of weights, whereas the "splintering" in Pugh's algorithm can grow with the coefficient values [23] (and thus exponentially in terms of input size, despite fixed dimension, unless there is some limit on coefficient sizes).

Vertex enumeration

Polyhedral libraries such as PolyLib and PPL exploit the double description of polyhedra and therefore naturally support vertex enumeration on (non-parametric) polytopes. The Omega Library internally performs vertex enumeration during the computation of the convex hull. PolyLib and isl provide vertex enumeration on parametric polytopes, which is essential for applying Barvinok's algorithm to parametric polytopes.

Indication of an approximate result

In some parts of a compiler, an approximate result is acceptable in certain cases. For example, when dependence analysis is used to guide loop transformation, it is generally acceptable to use a superset of the true dependencies—this can prevent an optimization but does not allow illegal code transformations. When the Omega Library produces an approximate answer, the answer is appropriately marked as an upper bound (e.g., via "and UNKNOWN") or a lower bound (e.g., via "or UNKNOWN"). Answers not marked this way are exact descriptions of sets of integer-valued points (except in cases of bugs in the software).

Handling nonlinear terms

When code contains a mixture of affine and non-affine terms, polyhedral libraries can, in principle, be used to produce approximate results, for example by simply omitting such terms when it is safe to do so. In addition to providing a way to flag such approximate results, the Omega Library allows restricted uses of "Uninterpreted Function Symbols" to stand in for any nonlinear term, providing a system that slightly improves the result of dependence analysis and (probably more significantly) provides a language for communication about these terms (to drive other analysis or communication with the programmer). Pugh and Wonnacott discussed a slightly less restricted domain than that allowed in the library, but this was never implemented (a description exists in

Wonnacott's dissertation).

Transitive closure operation

Some kinds of analysis, such as Pugh and Rosser's **iteration space slicing**, can be most easily stated in terms of the transitive closure of the dependence information. Both the Omega Library and isl provide a transitive closure operation that is exact for many cases that arise in programs with simple dependence patterns. In other cases, the Omega Library produces a subset of the transitive closure, while isl produces a superset. In the case of the Omega Library, the subset itself may be approximate, resulting in an upper bound (tagged) of a lower bound (not tagged) of the transitive closure. Note that the computation of an exact transitive closure is undecidable^[24].

References

- [1] <http://www.cs.umd.edu/projects/omega>
- [2] <http://www.chunchen.info/omega/>
- [3] <http://www.piplib.org/>
- [4] Paul Feautrier. *Parametric Integer Programming*. 1988
- [5] <http://icps.u-strasbg.fr/polylib/>
- [6] Doran K. Wilde. *A Library for Doing Polyhedral Operations*. 1993. tech report (<ftp://ftp.irisa.fr/techreports/1993/PI-785.ps.gz>)
- [7] <http://www.cs.unipr.it/ppl/>
- [8] <http://www.freshmeat.net/projects/isl>
- [9] Cedric Bastoul. *Code Generation in the Polyhedral Model Is Easier Than You Think*. PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques (2004)
- [10] <http://www.freshmeat.net/projects/barvinok>
- [11] Sebastian Pop, Albert Cohen , Cedric Bastoul , Sylvain Girbal, Pierre Jouvelot, Georges-Andr Silber et Nicolas Vasilache. *Graphite: Loop optimizations based on the polyhedral model for GCC*. 4th GCC Developer's Summit. Ottawa, Canada, June 2006.
- [12] <http://polly.grosser.es>
- [13] David Wonnacott. *Achieving Scalable Locality with Time Skewing*. International Journal of Parallel Programming 30.3 (2002)
- [14] David Wonnacott. *Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations*. (<http://doi.ieeecomputersociety.org/10.1109/IPDPS.2000.845979>) 14th International Parallel and Distributed Processing Symposium (IPDPS'00)
- [15] Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan. *Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model*. CC 2008 - International Conference on Compiler Construction
- [16] Yonghong Song, Zhiyuan Li. *New Tiling Techniques to Improve Cache Temporal Locality* (<http://portal.acm.org/citation.cfm?id=301618.301668&coll=portal&dl=ACM&type=series&idx=SERIES363&part=series&WantType=Proceedings&title=PLDI&CFID=36310011&CFTOKEN=72120731>). Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)
- [17] <http://www.cs.colostate.edu/~mstrout/>
- [18] <http://cs.haverford.edu/people/davew>
- [19] <http://www.math.lsa.umich.edu/~barvinok/>
- [20] William Pugh. *The Omega test: a fast and practical integer programming algorithm for dependence analysis* (<http://portal.acm.org/citation.cfm?id=125848>). Proceedings of the 1991 ACM/IEEE conference on Supercomputing
- [21] Robert Seater, David Wonnacott. *Polynomial Time Array Dataflow Analysis* (<http://www.springerlink.com/content/w2exje48bmv41b99/>), Languages and Compilers for Parallel Computing 2003
- [22] http://lipforge.ens-lyon.fr/tracker/index.php?func=detail&aid=3171&group_id=52&atid=296
- [23] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, Maurice Bruynooghe. *Counting Integer Points in Parametric Polytopes using Barvinok's Rational Functions* (<https://lirias.kuleuven.be/bitstream/123456789/124371/1/algoritmica.pdf>). Section 6.1 discusses Pugh's method and splintering.
- [24] Wayne Kelly, William Pugh, Evan Rosser, Tatiana Shpeisman. *Transitive Closure of Infinite Graphs and Its Applications*. Languages and Compilers for Parallel Computing, 8th International Workshop (LCPC 1995)

Case studies

GNU Compiler Collection

GNU Compiler Collection



Developer(s)	GNU Project
Initial release	May 23, 1987 ^[1]
Stable release	4.6.1 / June 27, 2011
Written in	C, C++
Operating system	Cross-platform
Platform	GNU
Type	Compiler
License	GNU General Public License (version 3 or later)
Website	http://gcc.gnu.org

The **GNU Compiler Collection (GCC)** is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux, the BSD family and Mac OS X.^[2] There is also an old (3.0) port of GCC to Plan9, running under its Ansi Posix Environment (APE).^[3]

GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in commercial, proprietary and closed source software development environments. GCC is also available for most embedded platforms, for example Symbian (called *gcce*),^[4] AMCC and Freescale Power Architecture-based chips.^[5] The compiler can target a wide variety of platforms, including videogame consoles such as the PlayStation 2^[6] and Dreamcast.^[7] Several companies^[8] make a business out of supplying and supporting GCC ports to various platforms, and chip manufacturers today consider a GCC port almost essential to the success of an architecture.

Originally named the **GNU C Compiler**, because it only handled the C programming language, GCC 1.0 was released in 1987, and the compiler was extended to compile C++ in December of that year.^[1] Front ends were later developed for Fortran, Pascal, Objective-C, Java, and Ada, among others.^[9]

The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

History

Richard Stallman started GCC in 1985. He extended an existing compiler to compile C. The compiler originally compiled Pastel, an extended, nonportable dialect of Pascal, and was written in Pastel. It was rewritten in C by Len Tower and Stallman,^[10] and released in 1987^[11] as the compiler for the GNU Project, in order to have a compiler available that was free software. Its development was supervised by the Free Software Foundation (FSF).^[12]

By 1991, GCC 1.x had reached a point of stability, but architectural limitations prevented many desired improvements, so the FSF started work on GCC 2.x.

As GCC was licensed under the GPL, programmers wanting to work in other directions—particularly those writing interfaces for languages other than C—were free to develop their own fork of the compiler (provided they meet the GPL's terms, including its requirements to distribute source code). Multiple forks proved inefficient and unwieldy, however, and the difficulty in getting work accepted by the official GCC project was greatly frustrating for many. The FSF kept such close control on what was added to the official version of GCC 2.x that GCC was used as one example of the "cathedral" development model in Eric S. Raymond's essay *The Cathedral and the Bazaar*.

With the release of 4.4BSD in 1994, GCC became the default compiler for most BSD systems.

EGCS fork

In 1997, a group of developers formed EGCS (Experimental/Enhanced GNU Compiler System),^[13] to merge several experimental forks into a single project. The basis of the merger was a GCC development snapshot taken between the 2.7 and 2.81 releases. Projects merged included g77 (FORTRAN), PGCC (P5 Pentium-optimized GCC), many C++ improvements, and many new architectures and operating system variants.^{[14] [15]}

EGCS development proved considerably more vigorous than GCC development, so much so that the FSF officially halted development on their GCC 2.x compiler, "blessed" EGCS as the official version of GCC and appointed the EGCS project as the GCC maintainers in April 1999. Furthermore, the project explicitly adopted the "bazaar" model over the "cathedral" model. With the release of GCC 2.95 in July 1999, the two projects were once again united.

GCC is now maintained by a varied group of programmers from around the world, under the direction of a steering committee.^[16] It has been ported to more kinds of processors and operating systems than any other compiler.^[17]

Development

GCC stable release

The current stable version of GCC is **4.6.1**, which was released on June 27, 2011.

GCC 4.6 supports many new Objective-C features, such as declared and synthesized properties, dot syntax, fast enumeration, optional protocol methods, method/protocol/class attributes, class extensions and a new GNU Objective-C runtime API. It also supports the Go programming language and includes the `libquadmath` library, which provides quadruple-precision mathematical functions on targets supporting the `__float128` datatype. The library is used to provide the `REAL(16)` type in GNU Fortran on such targets.

GCC uses many standard tools in its build, including Perl, Flex, Bison, and other common tools. In addition it currently requires three additional libraries to be present in order to build: GMP, MPC, and MPFR.

The previous major version, **4.5**, was initially released on April 14, 2010 (last minor version is 4.5.3, released on April 29, 2011). It included several minor new features (new targets, new language dialects) and a couple major new features:

- *Link-time optimization* optimizes across object file boundaries to directly improve the linked binary. Link-time optimization relies on an intermediate file containing the serialization of some -Gimple- representation included in the object file [18]. The file is generated alongside the object file during source compilation. Each source

compilation generates a separate object file and link-time helper file. When the object files are linked, the compiler is executed again and uses the helper files to optimize code across the separately compiled object files.

- *Plugins* can extend the GCC compiler directly [19]. Plugins allow a stock compiler to be tailored to specific needs by external code loaded as plugins. For example, plugins can add, replace, or even remove middle-end passes operating on *Gimple* representations. Several GCC plugins have already been published, notably:
 - TreeHydra^[20] to help with Mozilla code development
 - DragonEgg^[21] to use the GCC front-end with LLVM
 - MELT^[22] (site GCC MELT^[23]) to enable coding GCC extensions in a lisp-like domain-specific language providing powerful Pattern-matching
 - MILEPOST^[24] CTuning^[25] to use machine learning techniques to tune the compiler.

GCC trunk

The trunk concentrates the major part of the development efforts, where new features are implemented and tested. Eventually, the code from the trunk will become the next major release of GCC, with version **4.7**.

Uses

GCC is often chosen for developing software that is required to execute on a wide variety of hardware and/or operating systems. System-specific compilers provided by hardware or OS vendors can differ substantially, complicating both the software's source code and the scripts that invoke the compiler to build it. With GCC, most of the compiler is the same on every platform, so only code that explicitly uses platform-specific features must be rewritten for each system.

Languages

The standard compiler release 4.6 includes front ends for C (`gcc`), C++ (`g++`), Java (`gcj`), Ada (GNAT), Objective-C (`gobjc`), Objective-C++ (`gobjc++`) and Fortran (`gfortran`).^[26] Also available, but not in standard are Go (`gccgo`), Modula-2, Modula-3, Pascal (`gpc`), PL/I, D (`gdc`), Mercury, and VHDL (`ghdl`).^[27] A popular parallel language extension, OpenMP, is also supported.

The Fortran front end was `g77` before version 4.0, which only supports FORTRAN 77. In newer versions, `g77` is dropped in favor of the new `gfortran` front end that supports Fortran 95 and parts of Fortran 2003 as well.^[28] As the later Fortran standards incorporate the F77 standard, standards-compliant F77 code is also standards-compliant F90/95 code, and so can be compiled without trouble in `gfortran`. A front-end for CHILL was dropped due to a lack of maintenance.^[29]

A few experimental branches exist to support additional languages, such as the GCC UPC compiler^[30] for Unified Parallel C.

Architectures

GCC target processor families as of version 4.3 include:

- Alpha
- ARM
- Atmel AVR
- Blackfin
- HC12
- H8/300
- IA-32 (x86)
- IA-64

- Motorola 68000
- MIPS
- PA-RISC
- PDP-11
- PowerPC
- R8C/M16C/M32C
- SPU
- System/390/zSeries
- SuperH
- SPARC
- VAX
- x86-64

Lesser-known target processors supported in the standard release have included:

- 68HC11
- A29K
- ARC
- AVR32
- D30V
- DSP16xx
- ETRAX CRIS
- FR-30
- FR-V
- Intel i960
- IP2000
- M32R
- MCORE
- MIL-STD-1750A
- MMIX
- MN10200
- MN10300
- Motorola 88000
- NS32K
- ROMP
- Stormy16
- V850
- Xtensa

Additional processors have been supported by GCC versions maintained separately from the FSF version:

- Cortus APS3
- D10V
- EISC
- eSi-RISC
- Hexagon^[31]
- LatticeMico32
- LatticeMico8
- MeP
- MSP430^[32]

- MicroBlaze
- NEC SX architecture^[33]
- Nios II and Nios
- Motorola 6809
- MSP430
- OpenRISC 1200
- PDP-10
- PIC24/dsPIC
- System/370
- TIGCC (m68k variant)
- Z8000

The gcj Java compiler can target either a native machine language architecture or the Java Virtual Machine's Java bytecode.^[34] When retargeting GCC to a new platform, bootstrapping is often used.

Structure

GCC's external interface is generally standard for a UNIX compiler. Users invoke a driver program named `gcc`, which interprets command arguments, decides which language compilers to use for each input file, runs the assembler on their output, and then possibly runs the linker to produce a complete executable binary.

Each of the language compilers is a separate program that inputs source code and outputs assembly code. All have a common internal structure. A per-language front end parses the source code in that language and produces an abstract syntax tree ("tree" for short).

These are, if necessary, converted to the middle-end's input representation, called *GENERIC* form; the middle-end then gradually transforms the program towards its final form. Compiler optimizations and static code analysis techniques (such as FORTIFY_SOURCE,^[35] a compiler directive that attempts to discover some buffer overflows) are applied to the code. These work on multiple representations, mostly the architecture-independent GIMPLE representation and the architecture-dependent RTL representation. Finally, assembly language is produced using architecture-specific pattern matching originally based on an algorithm of Jack Davidson and Chris Fraser.

GCC is written primarily in C except for parts of the Ada front end. The distribution includes the standard libraries for Ada, C++, and Java whose code is mostly written in those languages.^[36] On some platforms, the distribution also includes a low-level runtime library, `libgcc`, written in a combination of machine-independent C and processor-specific assembly language, designed primarily to handle arithmetic operations that the target processor cannot perform directly.^[37]

In May 2010, the GCC steering committee decided to allow use of a C++ compiler to compile GCC.^[38] The compiler will be written in C plus a subset of features from C++. In particular, this was decided so that GCC's developers could use the "destructors" and "generics" features of C++.^[39]

Front-ends

Frontends vary internally, having to produce trees that can be handled by the backend. Currently, the parsers are all hand-coded recursive descent parsers, though there is no reason why a parser generator could not be used for new front-ends in the future (version 2 of the C compiler used a bison based grammar).

Until recently, the tree representation of the program was not fully independent of the processor being targeted.

The meaning of a tree was somewhat different for different language front-ends, and front-ends could provide their own tree codes. This was simplified with the introduction of *GENERIC* and *GIMPLE*, two new forms of language-independent trees that were introduced with the advent of GCC 4.0. *GENERIC* is more complex, based on the GCC 3.x Java front-end's intermediate representation. *GIMPLE* is a simplified *GENERIC*, in which various constructs are *lowered* to multiple *GIMPLE* instructions. The C, C++ and Java front ends produce *GENERIC* directly in the front end. Other front ends instead have different intermediate representations after parsing and convert these to *GENERIC*.

In either case, the so-called "gimplifier" then lowers this more complex form into the simpler SSA-based *GIMPLE* form that is the common language for a large number of new powerful language- and architecture-independent global (function scope) optimizations.

GENERIC and GIMPLE

GENERIC is an intermediate representation language used as a "middle-end" while compiling source code into executable binaries. A subset, called *GIMPLE*, is targeted by all the front-ends of GCC.

The middle stage of GCC does all the code analysis and optimization, working independently of both the compiled language and the target architecture, starting from the *GENERIC*^[40] representation and expanding it to Register Transfer Language. The *GENERIC* representation contains only the subset of the imperative programming constructs optimised by the middle-end.

In transforming the source code to *GIMPLE*^[41], complex expressions are split into a three address code using temporary variables. This representation was inspired by the SIMPLE representation proposed in the McCAT compiler^[42] by Laurie J. Hendren^[43] for simplifying the analysis and optimization of imperative programs.

Optimization

Optimization can occur during any phase of compilation, however the bulk of optimizations are performed after the syntax and semantic analysis of the front-end and before the code generation of the back-end, thus a common, even though somewhat contradictory, name for this part of the compiler is "middle end."

The exact set of GCC optimizations varies from release to release as it develops, but includes the standard algorithms, such as loop optimization, jump threading, common subexpression elimination, instruction scheduling, and so forth. The RTL optimizations are of less importance with the addition of global SSA-based optimizations on *GIMPLE* trees,^[44] as RTL optimizations have a much more limited scope, and have less high-level information.

Some of these optimizations performed at this level include dead code elimination, partial redundancy elimination, global value numbering, sparse conditional constant propagation, and scalar replacement of aggregates. Array dependence based optimizations such as automatic vectorization and automatic parallelization are also performed. Profile-guided optimization is also possible as demonstrated here: <http://gcc.gnu.org/install/build.html#TOC4>

Back-end

The behavior of GCC's back end is partly specified by preprocessor macros and functions specific to a target architecture, for instance to define the endianness, word size, and calling conventions. The front part of the back end uses these to help decide RTL generation, so although GCC's RTL is nominally processor-independent, the initial sequence of abstract instructions is already adapted to the target. At any moment, the actual RTL instructions forming the program representation have to comply with the machine description of the target architecture.

The machine description file contains RTL patterns, along with operand constraints, and code snippets to output the final assembly. The constraints indicate that a particular RTL pattern might only apply (for example) to certain hardware registers, or (for example) allow immediate operand offsets of only a limited size (*e.g.* 12, 16, 22, ... bit offsets, etc.). During RTL generation, the constraints for the given target architecture are checked. In order to issue a given snippet of RTL, it must match one (or more) of the RTL patterns in the machine description file, and satisfy the constraints for that pattern; otherwise, it would be impossible to convert the final RTL into assembly code.

Towards the end of compilation, valid RTL is reduced to a *strict* form in which each instruction refers to real machine registers and a pattern from the target's machine description file. Forming strict RTL is a complicated task; an important step is register allocation, where real, hardware registers are chosen to replace the initially-assigned pseudo-registers. This is followed by a "reloading" phase; any pseudo-registers that were not assigned a real hardware register are 'spilled' to the stack, and RTL to perform this spilling is generated. Likewise, offsets that are too large to fit in an actual instruction must be broken up and replaced by RTL sequences that will obey the offset constraints.

In the final phase the assembly code is built by calling a small snippet of code, associated with each pattern, to generate the real instructions from the target's instruction set, using the final registers, offsets and addresses chosen during the reload phase. The assembly-generation snippet may be just a string; in which case, a simple string substitution of the registers, offsets, and/or addresses into the string is performed. The assembly-generation snippet may also be a short block of C code, performing some additional work, but ultimately returning a string containing the valid assembly.

Compatible IDEs

Most integrated development environments written for GNU/Linux and some for other operating systems support GCC. These include:

- Anjuta
- Code::Blocks
- CodeLite
- Dev-C++
- Eclipse
- geany
- KDevelop
- NetBeans
- Qt Creator
- Xcode

Debugging GCC programs

The primary tool used to debug GCC code is the GNU Debugger (gdb). Among more specialized tools are Valgrind, for finding memory errors and leaks, and the graph profiler (gprof) that can determine how much time is spent in which routines, and how often they are called; this requires programs to be compiled with profiling options.

References

- [1] "GCC Releases" (<http://www.gnu.org/software/gcc/releases.html>). GNU Project. . Retrieved 2006-12-27.
- [2] "GNU Compiler Collection - Definition" (http://www.wordiq.com/definition/GNU_Compiler_Collection). <http://www.wordiq.com/>: wordiqQ. . Retrieved 2011-06-16. "It is the standard compiler for the open source Unix-like operating systems, and certain proprietary operating systems derived therefrom such as Mac OS X."
- [3] "/n/sources/extra/gcc" (<http://cm.bell-labs.com/sources/extra/gcc/>). http://plan9.bell-labs.com/wiki/plan9_porting_alien_software_to_plan_9/index.html: Bell Labs, Lucent. . Retrieved 2011-09-06.
- [4] "Symbian GCC Improvement Project" (<http://www.inf.u-szeged.hu/symbian-gcc/>). . Retrieved 2007-11-08.
- [5] "Linux Board Support Packages" (http://www.freescale.com/webapp/sps/site/overview.jsp?code=CW_BSP&fsrch=1). . Retrieved 2008-08-07.
- [6] "setting up gcc as a cross-compiler" (http://ps2stuff.playstation2-linux.com/gcc_build.html). *ps2stuff*. 2002-06-08. . Retrieved 2008-12-12.
- [7] "sh4 g++ guide" (<http://web.archive.org/web/20021220025554/http://www.ngine.de/gccguide.html>). Archived from the original (<http://www.ngine.de/gccguide.html>) on 2002-12-20. . Retrieved 2008-12-12. "This guide is intended for people who want to compile C++ code for their Dreamcast systems"
- [8] "FSF Service Directory" (<http://www.fsf.org/resources/service>). .
- [9] "Programming Languages Supported by GCC" (http://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/G_002b_002b-and-GCC.html#G_002b_002b-and-GCC). GNU Project. . Retrieved 2009-05-03.
- [10] Stallman, Richard M. (February 1986). "GNU Status" (<http://web.cecs.pdx.edu/~trent/gnu/bull/01/bull01.txt>). *GNU's Bulletin* (Free Software Foundation) 1 (1). . Retrieved 2006-09-26.
- [11] Tower, Leonard (1987) "GNU C compiler beta test release, (<http://groups.google.com/group/comp.lang.misc/msg/32eda22392c20f98>)" *comp.lang.misc* USENET newsgroup; see also <http://gcc.gnu.org/releases.html#timeline>
- [12] Stallman, Richard M. (2001) "Contributors to GCC, (http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_23.html#SEC260)" in *Using and Porting the GNU Compiler Collection (GCC)* (http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html#SEC_Top) for gcc version 2.95 (Cambridge, Mass.: Free Software Foundation)
- [13] "Pentium Compiler FAQ" (<http://home.schmorp.de/pgcc-faq.html#egcs>). .
- [14] "A Brief History of GCC" (<http://gcc.gnu.org/wiki/History>). .
- [15] "The Short History of GCC development" (http://www.softpanorama.org/People/Stallman/history_of_gcc_development.shtml). .
- [16] "GCC steering committee" (<http://gcc.gnu.org/steering.html>). .
- [17] Linux Information Project (<http://www.linfo.org/gcc.html>) (LINFO) accessed 2010-04-27
- [18] <http://gcc.gnu.org/wiki/LinkTimeOptimization>
- [19] <http://gcc.gnu.org/onlinedocs/gccint/Plugins.html>
- [20] <https://developer.mozilla.org/en/Treehydra>
- [21] <http://dragonegg.llvm.org/>
- [22] <http://gcc.gnu.org/wiki/MiddleEndLispTranslator>
- [23] <http://gcc-melt.org/>
- [24] <http://ctuning.org/wiki/index.php/CTools:MilepostGCC>
- [25] <http://ctuning.org/>
- [26] "gccgo language contribution accepted" (<http://article.gmane.org/gmane.comp.gcc.devel/111603>), gmane.org, Retrieved January 26, 2010.
- [27] GCC Front Ends (<http://gcc.gnu.org/frontends.html>), GCC.org, Retrieved May 11, 2008.
- [28] "Fortran 2003 Features in GNU Fortran" (<http://gcc.gnu.org/wiki/Fortran2003>). .
- [29] [PATCH] Remove chill (<http://gcc.gnu.org/ml/gcc-patches/2002-04/msg00887.html>), gcc.gnu.org, Retrieved July 29, 2010.
- [30] "GCC UPC (GCC Unified Parallel C) | intrepid.com" (<http://www.intrepid.com/upc.html>). intrepid.com<!. 2006-02-20. . Retrieved 2009-03-11.
- [31] "Hexagon Project Wiki" (<https://www.codeaurora.org/xwiki/bin/Hexagon/>). . "Hexagon dowload" (<https://www.codeaurora.org/patches/quic/hexagon/>). .
- [32] https://sourceforge.net/apps/mediawiki/mspgcc/index.php?title=MSPGCC_Wiki
- [33] "sx-gcc: port gcc to nec sx vector cpu" (<http://code.google.com/p/sx-gcc/>). .
- [34] "The GNU Compiler for the Java Programming Language" (<http://gcc.gnu.org/java/>). . Retrieved 2010-04-22.

- [35] "Security Features: Compile Time Buffer Checks (FORTIFY_SOURCE)" (<http://fedoraproject.org/wiki/Security/Features>). fedoraproject.org. . Retrieved 2009-03-11.
- [36] "languages used to make GCC" (<http://www.ohloh.net/projects/gcc/analyses/latest>). .
- [37] GCC Internals (<http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html>), GCC.org, Retrieved March 01, 2010.
- [38] "GCC allows C++ – to some degree" (<http://www.h-online.com/open/news/item/GCC-allows-C-to-some-degree-1012611.html>). The H. 1 June 2010. .
- [39] "An email by Richard Stallman on emacs-devel" (<http://lists.gnu.org/archive/html/emacs-devel/2010-07/msg00518.html>). . "The reason the GCC developers wanted to use it is for destructors and generics. These aren't much use in Emacs, which has GC and in which data types are handled at the Lisp level."
- [40] GENERIC (<http://gcc.gnu.org/onlinedocs/gccint/GENERIC.html>) in GNU Compiler Collection Internals
- [41] GIMPLE (<http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html>) in GNU Compiler Collection Internals
- [42] McCAT (<http://web.archive.org/web/20040812030043/www-acaps.cs.mcgill.ca/info/McCAT/McCAT.html>)
- [43] Laurie J. Hendren (<http://www.sable.mcgill.ca/~hendren/>)
- [44] From Source to Binary: The Inner Workings of GCC (<http://www.redhat.com/magazine/002dec04/features/gcc/>), by Diego Novillo, *Red Hat Magazine*, December 2004

Further reading

- Richard Stallman: *Using the GNU Compiler Collection (GCC)* (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/>), Free Software Foundation, 2008.
- Richard Stallman: *GNU Compiler Collection (GCC) Internals* (<http://gcc.gnu.org/onlinedocs/gccint/>), Free Software Foundation, 2008.
- Brian J. Gough: *An Introduction to GCC* (<http://www.network-theory.co.uk/gcc/intro/>), Network Theory Ltd., 2004 (Revised August 2005). ISBN 0-9541617-9-3.
- Arthur Griffith, *GCC: The Complete Reference*. McGrawHill/Osborne, 2002. ISBN 0-07-222405-3.

External links

- GCC homepage (<http://gcc.gnu.org/>)
- The official GCC manuals and user documentation (<http://gcc.gnu.org/onlinedocs/>), by the GCC developers
- Collection of GCC 4.0.2 architecture and internals documents (<http://www.cse.iitb.ac.in/grc/index.php?page=docs>) at I.I.T. Bombay.
- Kerner, Sean Michael (2006-03-02). "New GCC Heavy on Optimization" (<http://www.internetnews.com/dev-news/article.php/3588926>). internetnews.com.
- Kerner, Sean Michael (2005-04-22). "Open Source GCC 4.0: Older, Faster" (<http://www.internetnews.com/dev-news/article.php/3499881>). internetnews.com.
- From Source to Binary: The Inner Workings of GCC (<http://www.redhat.com/magazine/002dec04/features/gcc/>), by Diego Novillo, *Red Hat Magazine*, December 2004
- %20and%20GIMPLE.pdf A 2003 paper on GENERIC and GIMPLE (<ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC>)
- Marketing Cygnus Support (<http://www.toad.com/gnu/cygnus/index.html>), an essay covering GCC development for the 1990s, with 30 monthly reports for in the "Inside Cygnus Engineering" section near the end.
- EGCS 1.0 announcement (<http://www.goof.com/pcg/egcs.html>)
- EGCS 1.0 features list (<http://gcc.gnu.org/egcs-1.0/features.html>)
- Fear of Forking (http://linuxmafia.com/faq/Licensing_and_Law/forking.html), an essay by Rick Moen recording seven well-known forks, including the GCC/EGCS one
- A compiler course project (<http://www.cs.rochester.edu/twiki/bin/view/Main/ProjectHome>) based on GCC at the University of Rochester
- The stack-smashing protector (<http://www.trl.ibm.com/projects/security/ssp/>), a GCC extension

Java performance

Objectively comparing the **performance of a Java program** and another equivalent one written in another programming language such as C++ requires a carefully and thoughtfully constructed benchmark which compares programs expressing algorithms written in as identical a manner as technically possible. The target platform of Java's bytecode compiler is the Java platform, and the bytecode is either interpreted or compiled into machine code by the JVM. Other compilers almost always target a specific hardware and software platform, producing machine code that will stay virtually unchanged during its execution. Very different and hard-to-compare scenarios arise from these two different approaches: static vs. dynamic compilations and recompilations, the availability of precise information about the runtime environment and others.

The performance of the compiled Java program will depend on how smartly its particular tasks are going to be managed by the host JVM, and how well the JVM takes advantage of the features of the hardware and OS in doing so. Thus, any **Java performance** test or comparison has to always report the version, vendor, OS and hardware architecture of the used JVM. In a similar manner, the performance of the equivalent natively-compiled program will depend on the quality of its generated machine code, so the test or comparison also has to report the name, version and vendor of the used compiler, and its activated optimization directives.

Historically, Java programs' execution speed improved significantly due to the introduction of Just-In Time compilation (in 1997/1998 for Java 1.1),^[1] ^[2] ^[3] the addition of language features supporting better code analysis, and optimizations in the Java Virtual Machine itself (such as HotSpot becoming the default for Sun's JVM in 2000). Hardware execution of Java bytecode, such as that offered by ARM's Jazelle, can also offer significant performance improvements.

Virtual machine optimization techniques

Many optimizations have improved the performance of the Java Virtual Machine over time. However, although Java was often the first Virtual machine to implement them successfully, they have often been used in other similar platforms as well.

Just-In-Time compilation

Further information: Just-in-time compilation and HotSpot

Early Java Virtual Machines always interpreted bytecodes. This had a huge performance penalty (between a factor 10 and 20 for Java versus C in average applications).^[4] To combat this, a just-in-time (JIT) compiler was introduced into Java 1.1. Due to the high cost of compilation, an additional system called HotSpot was introduced into Java 1.2 and was made the default in Java 1.3. Using this framework, the Virtual Machine continually analyzes the program's performance for "hot spots" which are frequently or repeatedly executed. These are then targeted for optimization, leading to high performance execution with a minimum of overhead for less performance-critical code.^[5] ^[6] Some benchmarks show a 10-fold speed gain from this technique.^[7] However, due to time constraints, the compiler cannot fully optimize the program, and therefore the resulting program is slower than native code alternatives.^[8] ^[9]

Adaptive optimization

Further information: Adaptive optimization

Adaptive optimization is a technique in computer science that performs dynamic recompilation of portions of a program based on the current execution profile. With a simple implementation, an adaptive optimizer may simply make a trade-off between Just-in-time compilation and interpreting instructions. At another level, adaptive optimization may take advantage of local data conditions to optimize away branches and to use inline expansion.

A Virtual Machine like HotSpot is also able to deoptimize a previously JITed code. This allows it to perform aggressive (and potentially unsafe) optimizations, while still being able to deoptimize the code and fall back on a safe path later on.^[10] ^[11]

Garbage collection

Further information: Garbage collection (computer science)

The 1.0 and 1.1 Virtual Machines used a mark-sweep collector, which could fragment the heap after a garbage collection. Starting with Java 1.2, the Virtual Machines switched to a generational collector, which has a much better defragmentation behaviour.^[12] Modern Virtual Machines use a variety of techniques that have further improved the garbage collection performance.^[13]

Other optimization techniques

Split bytecode verification

Prior to executing a class, the Sun JVM verifies its bytecodes (see Bytecode verifier). This verification is performed lazily: classes bytecodes are only loaded and verified when the specific class is loaded and prepared for use, and not at the beginning of the program. (Note that other verifiers, such as the Java/400 verifier for IBM System i, can perform most verification in advance and cache verification information from one use of a class to the next.) However, as the Java Class libraries are also regular Java classes, they must also be loaded when they are used, which means that the start-up time of a Java program is often longer than for C++ programs, for example.

A technique named **Split-time verification**, first introduced in the J2ME of the Java platform, is used in the Java Virtual Machine since the Java version 6. It splits the verification of bytecode in two phases:^[14]

- Design-time - during the compilation of the class from source to bytecode
- runtime - when loading the class.

In practice this technique works by capturing knowledge that the Java compiler has of class flow and annotating the compiled method bytecodes with a synopsis of the class flow information. This does not make runtime verification appreciably less complex, but does allow some shortcuts.

Escape analysis and lock coarsening

Further information: Lock (computer science) and Escape analysis

Java is able to manage multithreading at the language level. Multithreading is a technique that allows programs to operate faster on computer systems that have multiple CPUs. Also, a multithreaded application has the ability to remain responsive to input, even when it is performing long running tasks.

However, programs that use multithreading need to take extra care of objects shared between threads, locking access to shared methods or blocks when they are used by one of the threads. Locking a block or an object is a time-consuming operation due to the nature of the underlying operating system-level operation involved (see concurrency control and lock granularity).

As the Java library does not know which methods will be used by more than one thread, the standard library always locks blocks when necessary in a multithreaded environment.

Prior to Java 6, the virtual machine always locked objects and blocks when asked to by the program even if there was no risk of an object being modified by two different threads at the same time. For example, in this case, a local `Vector` was locked before each of the `add` operations to ensure that it would not be modified by other threads (`Vector` is synchronized), but because it is strictly local to the method this is not necessary:

```
public String getNames() {  
    Vector v = new Vector();
```

```
v.add("Me");
v.add("You");
v.add("Her");
return v.toString();
}
```

Starting with Java 6, code blocks and objects are locked only when necessary,^[15] so in the above case, the virtual machine would not lock the Vector object at all.

As of version 6u14, Java includes experimental support for escape analysis.^[16]

Register allocation improvements

Prior to Java 6, allocation of registers was very primitive in the "client" virtual machine (they did not live across blocks), which was a problem in architectures which did not have a lot of registers available, such as x86. If there are no more registers available for an operation, the compiler must copy from register to memory (or memory to register), which takes time (registers are significantly faster to access). However the "server" virtual machine used a color-graph allocator and did not suffer from this problem.

An optimization of register allocation was introduced in Sun's JDK 6;^[17] it was then possible to use the same registers across blocks (when applicable), reducing accesses to the memory. This led to a reported performance gain of approximately 60% in some benchmarks.^[18]

Class data sharing

Class data sharing (called CDS by Sun) is a mechanism which reduces the startup time for Java applications, and also reduces memory footprint. When the JRE is installed, the installer loads a set of classes from the system jar file (the jar file containing all the Java class library, called rt.jar) into a private internal representation, and dumps that representation to a file, called a "shared archive". During subsequent JVM invocations, this shared archive is memory-mapped in, saving the cost of loading those classes and allowing much of the JVM's Metadata for these classes to be shared among multiple JVM processes.^[19]

The corresponding improvement for start-up time is more noticeable for small programs.^[20]

Sun Java versions performance improvements

Further information: Java version history

Apart from the improvements listed here, each Sun's Java version introduced many performance improvements in the Java API.

JDK 1.1.6 : First Just-in-time compilation (Symantec's JIT-compiler)^{[1] [3]}

J2SE 1.2 : Use of a generational collector.

J2SE 1.3 : Just-In-Time compilation by HotSpot.

J2SE 1.4 : See here^[21], for a Sun overview of performance improvements between 1.3 and 1.4 versions.

Java SE 5.0 : Class Data Sharing^[22]

Java SE 6 :

- Split bytecode verification
- Escape analysis and lock coarsening
- Register allocation Improvements

Other improvements:

- Java OpenGL Java 2D pipeline speed improvements^[23]
- Java 2D performance has also improved significantly in Java 6^[24]

See also 'Sun overview of performance improvements between Java 5 and Java 6'.^[25]

Java SE 6 Update 10

- Java Quick Starter reduces application start-up time by preloading part of JRE data at OS startup on disk cache.^[26]
- Parts of the platform that are necessary to execute an application accessed from the web when JRE is not installed are now downloaded first. The entire JRE is 12 MB, a typical Swing application only needs to download 4 MB to start. The remaining parts are then downloaded in the background.^[27]
- Graphics performance on Windows improved by extensively using Direct3D by default,^[28] and use Shaders on GPU to accelerate complex Java 2D operations.^[29]

Future improvements

Future performance improvements are planned for an update of Java 6 or Java 7:^[30]

- Provide JVM support for dynamic languages, following the prototyping work currently done on the Multi Language Virtual Machine,^[31]
- Enhance the existing concurrency library by managing parallel computing on multi-core processors,^{[32] [33]}
- Allow the virtual machine to use both the *Client* and *Server* compilers in the same session with a technique called *Tiered compilation*.^[34]
 - The *Client* would be used at startup (because it is good at startup and for small applications),
 - The *Server* would be used for long-term running of the application (because it outperforms the *Client* compiler for this).
- Replace the existing concurrent low-pause garbage collector (also called CMS or Concurrent Mark-Sweep collector) by a new collector called G1 (or Garbage First) to ensure consistent pauses over time.^{[35] [36]}

Comparison to other languages

Java is often Just-in-time compiled at runtime by the Java Virtual Machine, but may also be compiled ahead-of-time, just like C++. When Just-in-time compiled, its performance is generally:^[37]

- moderately slower than compiled languages such as C or C++,^[38]
- similar to other Just-in-time compiled languages such as C#,^[39]
- much faster than languages without an effective native-code compiler (JIT or AOT), such as Perl, Ruby, PHP and Python.^[40]

Program speed

Java is in some cases equal to C++ on low-level and numeric benchmarks.^[41]

Benchmarks often measure performance for small numerically-intensive programs. In some real-life programs, Java out-performs C. One example is the benchmark of Jake2 (a clone of Quake 2 written in Java by translating the original GPL C code). The Java 5.0 version performs better in some hardware configurations than its C counterpart.^[42] While it's not specified how the data was measured (for example if the original Quake 2 executable compiled in 1997 was used, which may be considered bad as current C compilers may achieve better optimizations for Quake), it notes how the same Java source code can have a huge speed boost just by updating the VM, something impossible to achieve with a 100% static approach. For other programs the C++ counterpart runs significantly faster than the Java equivalent^[43]

Some optimizations that are possible in Java and similar languages are not possible in C++:^[44]

- C-style pointers make optimization hard in languages that support them,

- The use of escape analysis techniques is limited in C++ for example, because the compiler does not know where an object will be used as accurately (also because of pointers).

The JVM is also able to perform processor specific optimizations or inline expansion. The ability to deoptimize code previously compiled or inlined allows to perform more aggressive optimizations than those performed with statically typed languages.^[45] ^[46]

Results for microbenchmarks between Java and C++ highly depend on which operations are compared. For example, when comparing with Java 5.0:

- 32 and 64 bits arithmetics operations,^[47] ^[48] File I/O^[49] and Exception handling,^[50] have a similar performance to comparable C++ programs
- Collections,^[51] ^[52] Heap allocation, as well as method calls^[53] are much better in Java than in C++.
- Arrays^[54] operations performance are better in C.
- Trigonometric functions performance is much better in C.^[55]

Multi-core performance

The scalability and performance of Java applications on multi-core systems is limited by the object allocation rate. This effect is sometimes called an "allocation wall".^[56] Also, applications that have not been tuned for multi-core systems may suffer from lock contention.^[57]

Startup time

Java startup time is often much slower than for C or C++, because a lot of classes (and first of all classes from the platform Class libraries) must be loaded before being used.

When compared against similar popular runtimes, for small programs running on a Windows machine, the startup time appears^[58] to be similar to Mono's and a little slower than .Net's.^[59]

It seems that much of the startup time is due to IO-bound operations rather than JVM initialization or class loading (the *rt.jar* class data file alone is 40 MB and the JVM must seek a lot of data in this huge file).^[26] Some tests showed that although the new Split bytecode verification technique improved class loading by roughly 40%, it only translated to about 5% startup improvement for large programs.^[60]

Albeit a small improvement it is more visible in small programs that perform a simple operation and then exit, because the Java platform data loading can represent many times the load of the actual program's operation.

Beginning with Java SE 6 Update 10, the Sun JRE comes with a Quick Starter that preloads class data at OS startup to get data from the disk cache rather than from the disk.

Excelsior JET approaches the problem from the other side. Its Startup Optimizer reduces the amount of data that must be read from the disk on application startup, and makes the reads more sequential.

Memory usage

Java memory usage is heavier than for C++, because:

- there is a 8-byte overhead for each object^[61] and 12-byte for each array^[62] in Java (32-bit; twice as much in 64-bit java). If size of an object is not a multiple of 8 bytes, it is rounded up to next multiple of 8. This means an object containing a single byte field occupies 16 bytes and requires 4-byte reference. However, C++ also allocates a pointer (usually 4 or 8 bytes) for every object that declares virtual functions.^[63]
- parts of the Java Library must be loaded prior to the program execution (at least the classes that are used "under the hood" by the program).^[64] This leads to a significant memory overhead^[65] for small applications when compared to its best known competitors Mono or .Net.
- both the Java binary and native recompilations will typically both be in memory

- the virtual machine itself consumes memory.
- in Java, a composite object (class A which uses instances of B and C) is created using references to allocated instances of B and C. In C++ the cost of the references can be avoided.
- lack of address arithmetic makes creating memory-efficient containers, such as tightly spaced structures and XOR linked lists, impossible.

However, it can be difficult to strictly compare the impact on memory of using Java versus C++. Some reasons why are:

- In C++, memory deallocation happens synchronously. In contrast, Java can do the deallocation asynchronously, possibly when the program is otherwise idle. A program that has periods of inactivity might perform better with Java because it is not deallocating memory during its active phase. However, memory deallocation is a very fast operation, especially in C++, where containers use allocators at a great extent. Thus garbage collector advantages are at best negligible.^[66]
- In C++, there can be a question of which part of the code "owns" an object, and is therefore responsible for deallocating it. For example, a container of objects might make copies of objects inserted into it, relying on the calling code to free its own copy, or it might insert the original object, creating an ambiguity of whether the calling code is handing the object off to the container (in which case the container should free the object when it is removed) or asking the container only to remember the object (in which case the calling code, not the container, will free the object later). For example, the C++ standard containers (in the STL) make copies of inserted objects.^[67] In Java, none of this is necessary because neither the calling code nor the container "owns" the object. So while the memory needed for a single object can be heavier than in C++, actual Java programs may create fewer objects, depending on the memory strategies of the C++ code, and if so, the time required for creating, copying, and deleting these objects is also not present in a Java program.

The consequences of these and other differences are highly dependent on the algorithms involved, the actual implementations of the memory allocation systems (**free**, **delete**, or the garbage collector), and the specific hardware. As a result, for applications in which memory is a critical factor of choosing between languages, a deep analysis is required.

One should also keep in mind that a program that uses a garbage collector needs about five times the memory of a program that uses explicit memory management in order to reach the same performance.^[68]

Trigonometric functions

Performance of trigonometric functions can be bad compared to C, because Java has strict specifications for the results of mathematical operations, which may not correspond to the underlying hardware implementation.^[69] On the x87, Java since 1.4 does argument reduction for sin and cos in software,^[70] causing a big performance hit for values outside the range.^[71]

Java Native Interface

The Java Native Interface has a high overhead associated with it, making it costly to cross the boundary between code running on the JVM and native code.^[72] ^[73] Java Native Access (JNA) provides Java programs easy access to native shared libraries (DLLs on Windows) without writing anything but Java code—no JNI or native code is required. This functionality is comparable to Windows' Platform/Invoke and Python's ctypes. Access is dynamic at runtime without code generation. But it comes with a cost and JNA is usually slower than JNI.^[74]

User interface

Swing has been perceived as slower than native widget toolkits, because it delegates the rendering of widgets to the pure Java Java 2D API. However, benchmarks comparing the performance of Swing versus the Standard Widget Toolkit, which delegates the rendering to the native GUI libraries of the operating system, show no clear winner, and the results greatly depend on the context and the environments.^[75]

Use for high performance computing

Recent independent studies seem to show that Java performance for high performance computing (HPC) is similar to Fortran on computation intensive benchmarks, but that JVMs still have scalability issues for performing intensive communication on a Grid Network.^[76]

However, high performance computing applications written in Java have recently won benchmark competitions. In 2008^[77] and 2009,^{[78] [79]} an Apache Hadoop (an open-source high performance computing project written in Java) based cluster was able to sort a terabyte and petabyte of integers the fastest. The hardware setup of the competing systems was not fixed, however.^{[80] [81]}

In programming contests

Programming contests provide by far the most objective known performance comparison, because contestants have equal time to implement their solution in language they know best, thus eliminating any bias against language they don't use, contests programs' output is checked and problems are diverse. As Java solutions run slower than solutions in other compiled languages,^{[82] [83]} it is not uncommon for online judges to use greater time limits for Java solutions^{[84] [85] [86]} to be fair to contestants using Java.

Notes

- [1] "Symantec's Just-In-Time Java Compiler To Be Integrated Into Sun JDK 1.1" (http://www.symantec.com/about/news/release/article.jsp?prid=19970407_03)..
- [2] "Apple Licenses Symantec's Just In Time (JIT) Compiler To Accelerate Mac OS Runtime For Java" (http://findarticles.com/p/articles/mi_hb6676/is_/ai_n26150624). .
- [3] "Java gets four times faster with new Symantec just-in-time compiler" (<http://www.infoworld.com/cgi-bin/displayStory.pl?980416.ehjdk.htm>). .
- [4] <http://www.shudo.net/jit/perf/>
- [5] Kawaguchi, Kohsuke (30 March 2008). "Deep dive into assembly code from Java" (http://weblogs.java.net/blog/kohsuke/archive/2008/03深深_进入.html). . Retrieved 2 April 2008.
- [6] "Fast, Effective Code Generation in a Just-In-Time Java Compiler" (<http://ei.cs.vt.edu/~cs5314/presentations/Group2PLDI.pdf>). Intel Corporation. . Retrieved 22 June 2007.
- [7] This article (<http://www.shudo.net/jit/perf/>) shows that the performance gain between interpreted mode and Hotspot amounts to more than a factor of 10.
- [8] Numeric performance in C, C# and Java (<http://www.itu.dk/~sestoft/papers/numericalperformance.pdf>)
- [9] Algorithmic Performance Comparison Between C, C++, Java and C# Programming Languages (<http://www.cherrystonesoftware.com/doc/AlgorithmicPerformance.pdf>)
- [10] "The Java HotSpot Virtual Machine, v1.4.1" (http://java.sun.com/products/spotdocs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_4.html#hotspot). Sun Microsystems. . Retrieved 20 April 2008.
- [11] Nutter, Charles (28 January 2008). "Lang.NET 2008: Day 1 Thoughts" (<http://headius.blogspot.com/2008/01/langnet-2008-day-1-thoughts.html>). . Retrieved 18 January 2011. "*Deoptimization is very exciting when dealing with performance concerns, since it means you can make much more aggressive optimizations...knowing you'll be able to fall back on a tried and true safe path later on*"
- [12] IBM DeveloperWorks Library (<http://www-128.ibm.com/developerworks/library/j-jtp01274.html>)
- [13] For example, the duration of pauses is less noticeable now. See for example this clone of Quake 2 written in Java: Jake2 (<http://bytonic.de/html/jake2.html>).
- [14] "New Java SE 6 Feature: Type Checking Verifier" (<https://jdk.dev.java.net/verifier.html>). Java.net. . Retrieved 18 January 2011.
- [15] (<http://www-128.ibm.com/developerworks/java/library/j-jtp10185/>)
- [16] (<http://java.dzone.com/articles/escape-analysis-java-6-update>)
- [17] Bug report: new register allocator, fixed in Mustang (JDK 6) b59 (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6320351)

- [18] Mustang's HotSpot Client gets 58% faster! (<http://weblogs.java.net/blog/2005/11/10/mustangs-hotspot-client-gets-58-faster>) in Osvaldo Pinali Doederlein's Blog at java.net
- [19] Class Data Sharing (<http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html>) at java.sun.com
- [20] Class Data Sharing in JDK 1.5.0 (<http://www.artima.com/forums/flat.jsp?forum=121&thread=56613>) in Java Buzz Forum at artima developer (<http://www.artima.com/>)
- [21] http://java.sun.com/j2se/1.4.2/performance_guide.html
- [22] Sun overview of performance improvements between 1.4 and 5.0 versions. (http://java.sun.com/performance/reference/whitepapers/5_0_performance.html)
- [23] STR-Crazier: Performance Improvements in Mustang (http://weblogs.java.net/blog/campbell/archive/2005/07/strcrazier_perf.html) in Chris Campbell's Blog at java.net
- [24] See here (http://jroller.com/page/dgilbert?entry=is_java_se_1_6) for a benchmark showing an approximately 60% performance boost from Java 5.0 to 6 for the application JFreeChart (<http://www.jfree.org>)
- [25] Java SE 6 Performance White Paper (http://java.sun.com/performance/reference/whitepapers/6_performance.html) at <http://java.sun.com>
- [26] Haase, Chet (May 2007). "Consumer JRE: Leaner, Meaner Java Technology" (<http://java.sun.com/developer/technicalArticles/javase/consumerjre#Quickstarter>). Sun Microsystems. . Retrieved 27 July 2007. *"At the OS level, all of these megabytes have to be read from disk, which is a very slow operation. Actually, it's the seek time of the disk that's the killer; reading large files sequentially is relatively fast, but seeking the bits that we actually need is not. So even though we only need a small fraction of the data in these large files for any particular application, the fact that we're seeking all over within the files means that there is plenty of disk activity."*
- [27] Haase, Chet (May 2007). "Consumer JRE: Leaner, Meaner Java Technology" (<http://java.sun.com/developer/technicalArticles/javase/consumerjre#JavaKernel>). Sun Microsystems. . Retrieved 27 July 2007.
- [28] Haase, Chet (May 2007). "Consumer JRE: Leaner, Meaner Java Technology" (<http://java.sun.com/developer/technicalArticles/javase/consumerjre#Performance>). Sun Microsystems. . Retrieved 27 July 2007.
- [29] Campbell, Chris (7 April 2007). "Faster Java 2D Via Shaders" (http://weblogs.java.net/blog/campbell/archive/2007/04/faster_java_2d.html). . Retrieved 18 January 2011.
- [30] Haase, Chet (May 2007). "Consumer JRE: Leaner, Meaner Java Technology" (<http://java.sun.com/developer/technicalArticles/javase/consumerjre>). Sun Microsystems. . Retrieved 27 July 2007.
- [31] "JSR 292: Supporting Dynamically Typed Languages on the Java Platform" (<http://www.jcp.org/en/jsr/detail?id=292>). jcp.org. . Retrieved 28 May 2008.
- [32] Goetz, Brian (4 March 2008). "Java theory and practice: Stick a fork in it, Part 2" (<http://www.ibm.com/developerworks/java/library/j-jtp03048.html?ca>). . Retrieved 9 March 2008.
- [33] Lorimer, R.J. (21 March 2008). "Parallelism with Fork/Join in Java 7" (http://www.infoq.com/news/2008/03/fork_join). infoq.com. . Retrieved 28 May 2008.
- [34] "New Compiler Optimizations in the Java HotSpot Virtual Machine" (<http://developers.sun.com/learning/javaoneonline/2006/coreplatform/TS-3412.pdf>). Sun Microsystems. May 2006. . Retrieved 30 May 2008.
- [35] Humble, Charles (13 May 2008). "JavaOne: Garbage First" (<http://www.infoq.com/news/2008/05/g1>). infoq.com. . Retrieved 7 September 2008.
- [36] Coward, Danny (12 November 2008). "Java VM: Trying a new Garbage Collector for JDK 7" (http://blogs.sun.com/theplanetarium/entry/java_vm_trying_a_new). . Retrieved 15 November 2008.
- [37] "Computer Language Benchmarks Game" (<http://shootout.alioth.debian.org/u32q/which-programming-languages-are-fastest.php?gcc=on&gpp=on&javasteady=on&java=on&csharp=on&javaxint=on&python3=on&python=on&jruby=on&php=on&perl=on&calc=chart>). shootout.alioth.debian.org. . Retrieved 2011-06-021.
- [38] "Computer Language Benchmarks Game" (<http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=java&lang2=gpp>). shootout.alioth.debian.org. . Retrieved 2011-06-021.
- [39] "Computer Language Benchmarks Game" (<http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=java&lang2=csharp>). shootout.alioth.debian.org. . Retrieved 2011-06-021.
- [40] "Computer Language Benchmarks Game" (<http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=java&lang2=python3>). shootout.alioth.debian.org. . Retrieved 2011-06-021.
- [41] Computer Language Benchmarks Game (<http://shootout.alioth.debian.org/u64q/java.php>)
- [42] : 260/250 frame/s versus 245 frame/s (see benchmark (<http://www.bytomic.de/html/benchmarks.html>))
- [43] Hundt, Robert. *Loop Recognition in C++/Java/Go/Scala* (<https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf>). Stanford, California: Scala Days 2011. . Retrieved 2 June 2011.
- [44] Lewis, J.P.; Neumann, Ulrich. "Performance of Java versus C++" (<http://scribblethink.org/Computer/javaCbenchmark.html>). Computer Graphics and Immersive Technology Lab, University of Southern California.
- [45] "The Java HotSpot Performance Engine: Method Inlining Example" (<http://java.sun.com/developer/technicalArticles/Networking/HotSpot/inlining.html>). Oracle Corporation.. Retrieved 2011-06-11.
- [46] Nutter, Charles (2008-05-03). "The Power of the JVM" (<http://blog.headius.com/2008/05/power-of-jvm.html>). . Retrieved 2011-06-11. *"What happens if you've already inlined A's method when B comes along? Here again the JVM shines. Because the JVM is essentially a dynamic language runtime under the covers, it remains ever-vigilant, watching for exactly these sorts of events to happen. And here's the*

really cool part: when situations change, the JVM can deoptimize. This is a crucial detail. Many other runtimes can only do their optimization once. C compilers must do it all ahead of time, during the build. Some allow you to profile your application and feed that into subsequent builds, but once you've released a piece of code it's essentially as optimized as it will ever get. Other VM-like systems like the CLR do have a JIT phase, but it happens early in execution (maybe before the system even starts executing) and doesn't ever happen again. The JVM's ability to deoptimize and return to interpretation gives it room to be optimistic...room to make ambitious guesses and gracefully fall back to a safe state, to try again later."

- [47] "Microbenchmarking C++, C#, and Java: 32-bit integer arithmetic" (<http://www.ddj.com/java/184401976?pgno=2>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [48] "Microbenchmarking C++, C#, and Java: 64-bit double arithmetic" (<http://www.ddj.com/java/184401976?pgno=12>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [49] "Microbenchmarking C++, C#, and Java: File I/O" (<http://www.ddj.com/java/184401976?pgno=15>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [50] "Microbenchmarking C++, C#, and Java: Exception" (<http://www.ddj.com/java/184401976?pgno=17>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [51] "Microbenchmarking C++, C#, and Java: Single Hash Map" (<http://www.ddj.com/java/184401976?pgno=18>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [52] "Microbenchmarking C++, C#, and Java: Multiple Hash Map" (<http://www.ddj.com/java/184401976?pgno=19>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [53] "Microbenchmarking C++, C#, and Java: Object creation/ destruction and method call" (<http://www.ddj.com/java/184401976?pgno=9>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [54] "Microbenchmarking C++, C#, and Java: Array" (<http://www.ddj.com/java/184401976?pgno=19>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [55] "Microbenchmarking C++, C#, and Java: Trigonometric functions" (<http://www.ddj.com/java/184401976?pgno=19>). Dr. Dobb's Journal. 1 July 2005. . Retrieved 18 January 2011.
- [56] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin and Ling Shao, Allocation wall: a limiting factor of Java applications on emerging multi-core platforms (<http://portal.acm.org/citation.cfm?id=1640116>), Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, 2009.
- [57] Kuo-Yi Chen, J. Morris Chang, Ting-Wei Hou, Multi-Threading in Java: Performance and Scalability on Multi-Core Systems (<http://home.engineering.iastate.edu/~morris/papers/10/ieetc10.pdf>), IEEE Transactions on Computers (<http://doi.ieeecomputersociety.org/10.1109/TC.2010.232>), 02 Dec. 2010. IEEE computer Society Digital Library. IEEE Computer Society.
- [58] <http://www.codeproject.com/KB/dotnet/RuntimePerformance.aspx#heading0010>
- [59] "Benchmark start-up and system performance for .Net, Mono, Java, C++ and their respective UI" (<http://www.codeproject.com/KB/dotnet/RuntimePerformance.aspx>). 2 September 2010..
- [60] "How fast is the new verifier?" (<http://forums.java.net/jive/thread.jspa?messageID=94530>). 7 February 2006. . Retrieved 9 May 2007.
- [61] http://java.sun.com/docs/books/performance/1st_edition/html/JPRAMFootprint.fm.html#24456
- [62] http://www.javamex.com/tutorials/memory/object_memory_usage.shtml
- [63] <http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=195>
- [64] <http://www.tommti-systems.de/go.html?http://www.tommti-systems.de/main-Dateien/reviews/languages/benchmarks.html>
- [65] <http://www.codeproject.com/KB/dotnet/RuntimePerformance.aspx>
- [66] <http://gcc.gnu.org/onlinedocs/libstdc++/manual/bk01pt04ch11.html#id462480>
- [67] <http://www.sgi.com/tech/stl/Container.html>
- [68] Matthew Hertz, Emery D. Berger, Quantifying the performance of garbage collection vs. explicit memory management (<http://www.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf>), Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 313 - 326, 2005.
- [69] "Math (Java Platform SE 6)" (<http://java.sun.com/javase/6/docs/api/java/lang/Math.html>). Sun Microsystems. . Retrieved 8 June 2008.
- [70] Gosling, James (27 July 2005). "Transcendental Meditation" (http://blogs.sun.com/jag/entry/transcendental_meditation). . Retrieved 8 June 2008.
- [71] W. Cowell-Shah, Christopher (8 January 2004). "Nine Language Performance Round-up: Benchmarking Math & File I/O" (<http://www.osnews.com/story/5602&page=3>). . Retrieved 8 June 2008.
- [72] Wilson, Steve; Jeff Kesselman (2001). "JavaTM Platform Performance: Using Native Code" (http://java.sun.com/docs/books/performance/1st_edition/html/JPNativeCode.fm.html). Sun Microsystems. . Retrieved 15 February 2008.
- [73] Kurzyniec, Dawid; Vaidy Sunderam. "Efficient Cooperation between Java and Native Codes - JNI Performance Benchmark" (<http://janet-project.sourceforge.net/papers/jnibench.pdf>). . Retrieved 15 February 2008.
- [74] "How does JNA performance compare to custom JNI?" (<https://jna.dev.java.net/#performance>). Sun Microsystems. . Retrieved 26 December 2009.
- [75] Igor, Križnar (10 May 2005). "SWT Vs. Swing Performance Comparison" (http://cosylib.cosylab.com/pub/CSS/DOC-SWT_Vs_Swing_Performance_Comparison.pdf). cosylib.com. . Retrieved 24 May 2008. "It is hard to give a rule-of-thumb where SWT would outperform Swing, or vice versa. In some environments (e.g., Windows), SWT is a winner. In others (Linux, VMware hosting Windows), Swing

and its redraw optimization outperform SWT significantly. Differences in performance are significant: factors of 2 and more are common, in either direction"

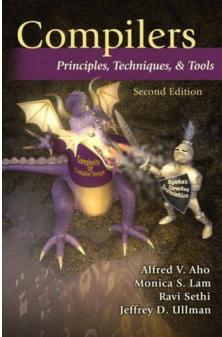
- [76] Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet, Guillermo L. Taboada (August 2008). "Current State of Java for HPC" (<http://hal.inria.fr/inria-00312039/en>). INRIA. . Retrieved 9 September 2008. "We first perform some micro benchmarks for various JVMs, showing the overall good performance for basic arithmetic operations(...). Comparing this implementation with a Fortran/MPI one, we show that they have similar performance on computation intensive benchmarks, but still have scalability issues when performing intensive communications."
- [77] Owen O'Malley - Yahoo! Grid Computing Team (July 2008). "Apache Hadoop Wins Terabyte Sort Benchmark" (http://developer.yahoo.net/blogs/hadoop/2008/07/apache_hadoop_wins_terabyte_sort_benchmark.html). . Retrieved 21 December 2008. "This is the first time that either a Java or an open source program has won."
- [78] "Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds" (http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html). CNET.com. 11 May 2009. . Retrieved 8 September 2010. "The hardware and operating system details are:(...)Sun Java JDK (1.6.0_05-b13 and 1.6.0_13-b03) (32 and 64 bit)"
- [79] "Hadoop breaks data-sorting world records" (http://news.cnet.com/8301-13846_3-10242392-62.html). CNET.com. 15 May 2009. . Retrieved 8 September 2010.
- [80] Chris Nyberg and Mehul Shah. "Sort Benchmark Home Page" (<http://sortbenchmark.org/>). . Retrieved 30 November 2010.
- [81] Czajkowski, Grzegorz (21 November 2008). "Sorting 1PB with MapReduce" (<http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html>). google. . Retrieved 1 December 2010.
- [82] <http://topcoder.com/home/tco10/2010/06/08/algorithms-problem-writing/>
- [83] <http://acm.timus.ru/help.aspx?topic=java&locale=en>
- [84] <http://acm.pku.edu.cn/JudgeOnline/faq.htm#q11>
- [85] <http://acm.tju.edu.cn/toj/faq.html#qj>
- [86] <http://m-judge.maximum.vc/faq.cgi#tl>

External links

- Site dedicated to Java performance information (<http://javaperformancetuning.com/>)
- Debugging Java performance problems (<http://prefetch.net/presentations/DebuggingJavaPerformance.pdf>)
- Sun's Java performance portal (<http://java.sun.com/docs/performance/>)

Literature

Compilers: Principles, Techniques, and Tools

<i>Compilers: Principles, Techniques, and Tools</i>	
	
The cover of the second edition (North American), showing a knight and dragon	
Author(s)	Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
Language	English
Publisher	Pearson Education, Inc
Publication date	1986, 2006
ISBN	ISBN 0-201-10088-6, ISBN 0-321-48681-1
OCLC Number	12285707 [1]
Dewey Decimal	005.4/53 19
LC Classification	QA76.76.C65 A37 1986

Compilers: Principles, Techniques, and Tools^[2] is a famous computer science textbook by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman about compiler construction. Although more than two decades have passed since the publication of the first edition, it is widely regarded as the classic definitive compiler technology text.^[3]

It is known as the **Dragon Book** because its covers depict a knight and a dragon in battle, a metaphor for conquering complexity. This name can also refer to Aho and Ullman's older *Principles of Compiler Design*.

First edition

The first edition is informally called the “red dragon book” to distinguish it from the second edition and from Aho & Ullman’s *Principles of Compiler Design* (1977, sometimes known as the “green dragon book” because the dragon on its cover is green).

A new edition of the book was published in August 2006.

Topics covered in the first edition include:

- Compiler structure
- Lexical analysis (including regular expressions and finite automata)
- Syntax analysis (including context-free grammars, LL parsers, bottom-up parsers, and LR parsers)
- Syntax-directed translation

- Type checking (including type conversions and polymorphism)
- Run-time environment (including parameter passing, symbol tables, and storage allocation)
- Code generation (including intermediate code generation)
- Code optimization

Second edition

Following in the tradition of its two predecessors, the second edition features a dragon and a knight on its cover; for this reason, the series of books is commonly known as the *Dragon Books*. Different editions in the series are further distinguished by the color of the dragon. This edition is informally known as the **purple dragon**. Monica S. Lam of Stanford University became a co-author with this edition.

The second edition includes several additional topics that are not covered in the first edition. New topics include

- directed translation
- new data flow analyses
- parallel machines
- JIT compiling
- garbage collection
- new case studies.

References

- [1] <http://worldcat.org/oclc/12285707>
- [2] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986. ISBN 0-201-10088-6
- [3] "The Top 9 1/2 Books in a Hacker's Bookshelf" (<http://grokcode.com/11/the-top-9-in-a-hackers-bookshelf/>). . Retrieved 2010-10-23.

External links

- Book Website at Stanford with link to Errata (<http://dragonbook.stanford.edu/>)
- Sample chapters from the second edition. (http://wps.aw.com/aw_aho_compilers_2/0,11227,2663889,-00.html)
- The 2006 edition: ISBN 0-321-48681-1

Principles of Compiler Design

Principles of Compiler Design, by Alfred Aho and Jeffrey D. Ullman, is a classic textbook on compilers for computer programming languages.

It is often called the "dragon book" because its cover depicts a knight and a dragon in battle; the dragon is green, and labelled "Complexity of Compiler Construction", while the knight wields a lance labeled "LALR parser generator". The book may be called the "green dragon book" to distinguish it from its successor, Aho, Sethi & Ullman's *Compilers: Principles, Techniques, and Tools*, which is the "red dragon book" because the dragon on its cover is red. The second edition of *Compilers: Principles, Techniques, and Tools* added a fourth author, Monica S. Lam, and the dragon became purple; hence becoming the "purple dragon book."

The back cover offers a humorously different viewpoint on the problem - the dragon is replaced by windmills, and the knight is Don Quixote.

Principles of Compiler Design is now rather dated, but when it came out in 1977, it was hailed for its practical bent; it included treatments of all compilation phases, with sufficient algorithmic detail that students could build their own small compilers in a semester.

The book was published by Addison-Wesley, ISBN 0-201-00022-9. The acknowledgments mention that the book was entirely typeset at Bell Labs using troff on the Unix operating system, which at that time had been little seen outside the Labs.

The Design of an Optimizing Compiler

The Design of an Optimizing Compiler (Elsevier Science Ltd, 1980, ISBN 0444001581), by William Wulf, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke, was published in 1975 by Elsevier. It describes the BLISS compiler for the PDP-11, written at Carnegie Mellon University in the early 1970s. The compiler ran on a PDP-10 and was one of the first to produce well-optimized code for a minicomputer. Because of its elegant design and the quality of the generated code, the compiler and book remain classics in the compiler field.

Although the original book has been out of print for many years, a print on demand version remains available from University Microfilms International.

Article Sources and Contributors

Compiler construction *Source:* <http://en.wikipedia.org/w/index.php?oldid=441884221> *Contributors:* -

Compiler *Source:* <http://en.wikipedia.org/w/index.php?oldid=451085484> *Contributors:* -

Interpreter *Source:* <http://en.wikipedia.org/w/index.php?oldid=450426290> *Contributors:* -

History of compiler writing *Source:* <http://en.wikipedia.org/w/index.php?oldid=410767109> *Contributors:* -

Lexical analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=450960372> *Contributors:* -

Regular expression *Source:* <http://en.wikipedia.org/w/index.php?oldid=451308609> *Contributors:* -

Regular expression examples *Source:* <http://en.wikipedia.org/w/index.php?oldid=444061404> *Contributors:* -

Finite-state machine *Source:* <http://en.wikipedia.org/w/index.php?oldid=450288011> *Contributors:* -

Preprocessor *Source:* <http://en.wikipedia.org/w/index.php?oldid=445864877> *Contributors:* -

Parsing *Source:* <http://en.wikipedia.org/w/index.php?oldid=450496028> *Contributors:* -

Lookahead *Source:* <http://en.wikipedia.org/w/index.php?oldid=445931095> *Contributors:* -

Symbol table *Source:* <http://en.wikipedia.org/w/index.php?oldid=450361755> *Contributors:* -

Abstract syntax *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638013> *Contributors:* -

Abstract syntax tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=450919292> *Contributors:* -

Context-free grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=448110869> *Contributors:* -

Terminal and nonterminal symbols *Source:* <http://en.wikipedia.org/w/index.php?oldid=440691085> *Contributors:* -

Left recursion *Source:* <http://en.wikipedia.org/w/index.php?oldid=431009368> *Contributors:* -

Backus–Naur Form *Source:* <http://en.wikipedia.org/w/index.php?oldid=450959405> *Contributors:* -

Extended Backus–Naur Form *Source:* <http://en.wikipedia.org/w/index.php?oldid=438156237> *Contributors:* -

TBNF *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638406> *Contributors:* -

Top-down parsing *Source:* <http://en.wikipedia.org/w/index.php?oldid=446794692> *Contributors:* -

Recursive descent parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=445532439> *Contributors:* -

Tail recursive parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=352279529> *Contributors:* -

Parsing expression grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=450776820> *Contributors:* -

LL parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=450997243> *Contributors:* -

LR parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=450717926> *Contributors:* -

Parsing table *Source:* <http://en.wikipedia.org/w/index.php?oldid=444041537> *Contributors:* -

Simple LR parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=428214353> *Contributors:* -

Canonical LR parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=413201666> *Contributors:* -

GLR parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=431897149> *Contributors:* -

LALR parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=451016219> *Contributors:* -

Recursive ascent parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=425836816> *Contributors:* -

Parser combinator *Source:* <http://en.wikipedia.org/w/index.php?oldid=450063175> *Contributors:* -

Bottom-up parsing *Source:* <http://en.wikipedia.org/w/index.php?oldid=445414668> *Contributors:* -

Chomsky normal form *Source:* <http://en.wikipedia.org/w/index.php?oldid=448796137> *Contributors:* -

CYK algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=451521312> *Contributors:* -

Simple precedence grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=346482459> *Contributors:* -

Simple precedence parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=391103229> *Contributors:* -

Operator-precedence grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=428207078> *Contributors:* -

Operator-precedence parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=439971707> *Contributors:* -

Shunting-yard algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=450298906> *Contributors:* -

Chart parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=371677735> *Contributors:* -

Earley parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=440152782> *Contributors:* -

The lexer hack *Source:* <http://en.wikipedia.org/w/index.php?oldid=409387145> *Contributors:* -

Scannerless parsing *Source:* <http://en.wikipedia.org/w/index.php?oldid=417257703> *Contributors:* -

Attribute grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=430251797> *Contributors:* -

L-attributed grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638235> *Contributors:* -

LR-attributed grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=415080280> *Contributors:* -

S-attributed grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=422254483> *Contributors:* -

ECLR-attributed grammar *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638197> *Contributors:* -

Intermediate representation *Source:* <http://en.wikipedia.org/w/index.php?oldid=432011342> *Contributors:* -

Intermediate language *Source:* <http://en.wikipedia.org/w/index.php?oldid=445179409> *Contributors:* -

Control flow graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=450323533> *Contributors:* -

Basic block *Source:* <http://en.wikipedia.org/w/index.php?oldid=449660547> *Contributors:* -

Call graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=444305974> *Contributors:* -

Data-flow analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=438845742> *Contributors:* -

Use-define chain *Source:* <http://en.wikipedia.org/w/index.php?oldid=425972685> *Contributors:* -

Live variable analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=426956206> *Contributors:* -

Reaching definition *Source:* <http://en.wikipedia.org/w/index.php?oldid=422520656> *Contributors:* -

Three address code *Source:* <http://en.wikipedia.org/w/index.php?oldid=451144677> *Contributors:* -

Static single assignment form *Source:* <http://en.wikipedia.org/w/index.php?oldid=450653179> *Contributors:* -

Dominator *Source:* <http://en.wikipedia.org/w/index.php?oldid=435620107> *Contributors:* -

C3 linearization *Source:* <http://en.wikipedia.org/w/index.php?oldid=432031969> *Contributors:* -

Intrinsic function *Source:* <http://en.wikipedia.org/w/index.php?oldid=422464509> *Contributors:* -

Aliasing *Source:* <http://en.wikipedia.org/w/index.php?oldid=435183209> *Contributors:* -

Alias analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=450267473> *Contributors:* -

Array access analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=419149424> *Contributors:* -

Pointer analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=411157612> *Contributors:* -

Escape analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=446806396> *Contributors:* -

Shape analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=410758714> *Contributors:* -

Loop dependence analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638268> *Contributors:* -

Program slicing *Source:* <http://en.wikipedia.org/w/index.php?oldid=430361869> *Contributors:* -

Compiler optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=447103826> *Contributors:* -

Peephole optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=404686923> *Contributors:* -

Copy propagation *Source:* <http://en.wikipedia.org/w/index.php?oldid=429993596> *Contributors:* -

Constant folding *Source:* <http://en.wikipedia.org/w/index.php?oldid=449214296> *Contributors:* -

Sparse conditional constant propagation *Source:* <http://en.wikipedia.org/w/index.php?oldid=412017419> *Contributors:* -

Common subexpression elimination *Source:* <http://en.wikipedia.org/w/index.php?oldid=450334654> *Contributors:* -

Partial redundancy elimination *Source:* <http://en.wikipedia.org/w/index.php?oldid=398570191> *Contributors:* -

Global value numbering *Source:* <http://en.wikipedia.org/w/index.php?oldid=435619067> *Contributors:* -

Strength reduction *Source:* <http://en.wikipedia.org/w/index.php?oldid=450717519> *Contributors:* -

Bounds-checking elimination *Source:* <http://en.wikipedia.org/w/index.php?oldid=404110851> *Contributors:* -

Inline expansion *Source:* <http://en.wikipedia.org/w/index.php?oldid=449512722> *Contributors:* -

Return value optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=447456195> *Contributors:* -

Dead code *Source:* <http://en.wikipedia.org/w/index.php?oldid=436869442> *Contributors:* -

Dead code elimination *Source:* <http://en.wikipedia.org/w/index.php?oldid=436037118> *Contributors:* -

Unreachable code *Source:* <http://en.wikipedia.org/w/index.php?oldid=417932367> *Contributors:* -

Redundant code *Source:* <http://en.wikipedia.org/w/index.php?oldid=437407020> *Contributors:* -

Jump threading *Source:* <http://en.wikipedia.org/w/index.php?oldid=449807195> *Contributors:* -

Superoptimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=424505323> *Contributors:* -

Loop optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=433085104> *Contributors:* -

Induction variable *Source:* <http://en.wikipedia.org/w/index.php?oldid=450323128> *Contributors:* -

Loop fission *Source:* <http://en.wikipedia.org/w/index.php?oldid=434640532> *Contributors:* -

Loop fusion *Source:* <http://en.wikipedia.org/w/index.php?oldid=350534333> *Contributors:* -

Loop inversion *Source:* <http://en.wikipedia.org/w/index.php?oldid=332070324> *Contributors:* -

Loop interchange *Source:* <http://en.wikipedia.org/w/index.php?oldid=395057618> *Contributors:* -

Loop-invariant code motion *Source:* <http://en.wikipedia.org/w/index.php?oldid=450007727> *Contributors:* -

Loop nest optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=444722594> *Contributors:* -

Manifest expression *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638276> *Contributors:* -

Polytope model *Source:* <http://en.wikipedia.org/w/index.php?oldid=431244126> *Contributors:* -

Loop unwinding *Source:* <http://en.wikipedia.org/w/index.php?oldid=447687366> *Contributors:* -

Loop splitting *Source:* <http://en.wikipedia.org/w/index.php?oldid=422203524> *Contributors:* -

Loop tiling *Source:* <http://en.wikipedia.org/w/index.php?oldid=394090334> *Contributors:* -

Loop unswitching *Source:* <http://en.wikipedia.org/w/index.php?oldid=330204327> *Contributors:* -

Interprocedural optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=440213241> *Contributors:* -

Whole program optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=398872309> *Contributors:* -

Adaptive optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=432940462> *Contributors:* -

Lazy evaluation *Source:* <http://en.wikipedia.org/w/index.php?oldid=447380738> *Contributors:* -

Partial evaluation *Source:* <http://en.wikipedia.org/w/index.php?oldid=445602948> *Contributors:* -

Profile-guided optimization *Source:* <http://en.wikipedia.org/w/index.php?oldid=446176895> *Contributors:* -

Automatic parallelization *Source:* <http://en.wikipedia.org/w/index.php?oldid=447838611> *Contributors:* -

Loop scheduling *Source:* <http://en.wikipedia.org/w/index.php?oldid=189288367> *Contributors:* -

Vectorization *Source:* <http://en.wikipedia.org/w/index.php?oldid=438871271> *Contributors:* -

Superword Level Parallelism *Source:* <http://en.wikipedia.org/w/index.php?oldid=272045216> *Contributors:* -

Code generation *Source:* <http://en.wikipedia.org/w/index.php?oldid=450996388> *Contributors:* -

Name mangling *Source:* <http://en.wikipedia.org/w/index.php?oldid=451040114> *Contributors:* -

Register allocation *Source:* <http://en.wikipedia.org/w/index.php?oldid=450282965> *Contributors:* -

Chaitin's algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=150879119> *Contributors:* -

Rematerialization *Source:* <http://en.wikipedia.org/w/index.php?oldid=255588519> *Contributors:* -

Sethi-Ullman algorithm *Source:* <http://en.wikipedia.org/w/index.php?oldid=410574819> *Contributors:* -

Data structure alignment *Source:* <http://en.wikipedia.org/w/index.php?oldid=435004317> *Contributors:* -

Instruction selection *Source:* <http://en.wikipedia.org/w/index.php?oldid=402800791> *Contributors:* -

Instruction scheduling *Source:* <http://en.wikipedia.org/w/index.php?oldid=442992323> *Contributors:* -

Software pipelining *Source:* <http://en.wikipedia.org/w/index.php?oldid=441030193> *Contributors:* -

Trace scheduling *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638428> *Contributors:* -

Just-in-time compilation *Source:* <http://en.wikipedia.org/w/index.php?oldid=451482731> *Contributors:* -

Bytecode *Source:* <http://en.wikipedia.org/w/index.php?oldid=449255442> *Contributors:* -

Dynamic compilation *Source:* <http://en.wikipedia.org/w/index.php?oldid=433550219> *Contributors:* -

Dynamic recompilation *Source:* <http://en.wikipedia.org/w/index.php?oldid=444744577> *Contributors:* -

Object file *Source:* <http://en.wikipedia.org/w/index.php?oldid=450967366> *Contributors:* -

Code segment *Source:* <http://en.wikipedia.org/w/index.php?oldid=444812132> *Contributors:* -

Data segment *Source:* <http://en.wikipedia.org/w/index.php?oldid=451257734> *Contributors:* -

.bss *Source:* <http://en.wikipedia.org/w/index.php?oldid=451251289> *Contributors:* -

Literal pool *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638257> *Contributors:* -

Overhead code *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638331> *Contributors:* -

Link time *Source:* <http://en.wikipedia.org/w/index.php?oldid=428889474> *Contributors:* -

Relocation *Source:* <http://en.wikipedia.org/w/index.php?oldid=450779343> *Contributors:* -

Library *Source:* <http://en.wikipedia.org/w/index.php?oldid=448772275> *Contributors:* -

Static build *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638381> *Contributors:* -

Architecture Neutral Distribution Format *Source:* <http://en.wikipedia.org/w/index.php?oldid=448470798> *Contributors:* -

Bootstrapping *Source:* <http://en.wikipedia.org/w/index.php?oldid=431167440> *Contributors:* -

Compiler correctness *Source:* <http://en.wikipedia.org/w/index.php?oldid=450420910> *Contributors:* -

Jensen's Device *Source:* <http://en.wikipedia.org/w/index.php?oldid=426547289> *Contributors:* -

Man or boy test *Source:* <http://en.wikipedia.org/w/index.php?oldid=448693863> *Contributors:* -

Cross compiler *Source:* <http://en.wikipedia.org/w/index.php?oldid=450500324> *Contributors:* -

Source-to-source compiler *Source:* <http://en.wikipedia.org/w/index.php?oldid=447142266> *Contributors:* -

Compiler-compiler *Source:* <http://en.wikipedia.org/w/index.php?oldid=442076560> *Contributors:* -

PQCC *Source:* <http://en.wikipedia.org/w/index.php?oldid=434237481> *Contributors:* -

Compiler Description Language *Source:* <http://en.wikipedia.org/w/index.php?oldid=412638125> *Contributors:* -

Comparison of regular expression engines *Source:* <http://en.wikipedia.org/w/index.php?oldid=451509140> *Contributors:* -

Comparison of parser generators *Source:* <http://en.wikipedia.org/w/index.php?oldid=451353443> *Contributors:* -

Lex *Source:* <http://en.wikipedia.org/w/index.php?oldid=445548323> *Contributors:* -

flex lexical analyser *Source:* <http://en.wikipedia.org/w/index.php?oldid=445774838> *Contributors:* -

Quex *Source:* <http://en.wikipedia.org/w/index.php?oldid=444925080> *Contributors:* -

JLex *Source:* <http://en.wikipedia.org/w/index.php?oldid=320088244> *Contributors:* -

Ragel *Source:* <http://en.wikipedia.org/w/index.php?oldid=441378828> *Contributors:* -

yacc *Source:* <http://en.wikipedia.org/w/index.php?oldid=449524668> *Contributors:* -

Berkeley Yacc *Source:* <http://en.wikipedia.org/w/index.php?oldid=448632178> *Contributors:* -

ANTLR *Source:* <http://en.wikipedia.org/w/index.php?oldid=448619037> *Contributors:* -

GNU bison *Source:* <http://en.wikipedia.org/w/index.php?oldid=450255589> *Contributors:* -

Coco/R *Source:* <http://en.wikipedia.org/w/index.php?oldid=450254892> *Contributors:* -

GOLD *Source:* <http://en.wikipedia.org/w/index.php?oldid=403976607> *Contributors:* -

JavaCC *Source:* <http://en.wikipedia.org/w/index.php?oldid=448068427> *Contributors:* -

JetPAG *Source:* <http://en.wikipedia.org/w/index.php?oldid=411413597> *Contributors:* -

Lemon Parser Generator *Source:* <http://en.wikipedia.org/w/index.php?oldid=360991332> *Contributors:* -

ROSE compiler framework *Source:* <http://en.wikipedia.org/w/index.php?oldid=432098876> *Contributors:* -

SableCC *Source:* <http://en.wikipedia.org/w/index.php?oldid=429170183> *Contributors:* -

Scannerless Boolean Parser *Source:* <http://en.wikipedia.org/w/index.php?oldid=360198296> *Contributors:* -

Spirit Parser Framework *Source:* <http://en.wikipedia.org/w/index.php?oldid=439988281> *Contributors:* -

S/SL programming language *Source:* <http://en.wikipedia.org/w/index.php?oldid=445865446> *Contributors:* -

SYNTAX *Source:* <http://en.wikipedia.org/w/index.php?oldid=419386733> *Contributors:* -

Syntax Definition Formalism *Source:* <http://en.wikipedia.org/w/index.php?oldid=411874854> *Contributors:* -

TREE-META *Source:* <http://en.wikipedia.org/w/index.php?oldid=435580112> *Contributors:* -

Frameworks supporting the polyhedral model *Source:* <http://en.wikipedia.org/w/index.php?oldid=450332083> *Contributors:* -

GNU Compiler Collection *Source:* <http://en.wikipedia.org/w/index.php?oldid=450661525> *Contributors:* -

Java performance *Source:* <http://en.wikipedia.org/w/index.php?oldid=447275111> *Contributors:* -

Compilers: Principles, Techniques, and Tools *Source:* <http://en.wikipedia.org/w/index.php?oldid=412167231> *Contributors:* -

Principles of Compiler Design *Source:* <http://en.wikipedia.org/w/index.php?oldid=430002799> *Contributors:* -

The Design of an Optimizing Compiler *Source:* <http://en.wikipedia.org/w/index.php?oldid=433177272> *Contributors:* -

Image Sources, Licenses and Contributors

Image:Compiler.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Compiler.svg> *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Surachit

File:UML state machine Fig5.png *Source:* http://en.wikipedia.org/w/index.php?title=File:UML_state_machine_Fig5.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Mirosamek (talk)

Image:SdlStateMachine.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:SdlStateMachine.png> *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* -

File:Finite state machine example with comments.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Finite_state_machine_example_with_comments.svg *License:* Public Domain *Contributors:* Macguy314

File:Fsm parsing word nice.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Fsm_parsing_word_nice.svg *License:* Public Domain *Contributors:* en:User:Thowa, redrawn by User:Stammered

File:DFAexample.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:DFAexample.svg> *License:* Public Domain *Contributors:* Cepheus

File:Fsm mealy model door control.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Fsm_mealy_model_door_control.jpg *License:* Public Domain *Contributors:* Original uploader was Thowa at en.wikipedia

File:Finite State Machine Logic.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Finite_State_Machine_Logic.svg *License:* Public Domain *Contributors:* jjbeard

File:4 bit counter.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:4_bit_counter.svg *License:* Public Domain *Contributors:* Gargan

Image:Parser Flow.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Parser_Flow.gif *License:* Public Domain *Contributors:* -

File:Abstract syntax tree for Euclidean algorithm.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Abstract_syntax_tree_for_Euclidean_algorithm.svg *License:* Creative Commons Zero *Contributors:* Dcoetze

Image:LR Parser.png *Source:* http://en.wikipedia.org/w/index.php?title=File:LR_Parser.png *License:* GNU Free Documentation License *Contributors:* -

Image:parseTree.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:ParseTree.svg> *License:* Public Domain *Contributors:* Traced by User:Stammered

File:Shunting yard.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Shunting_yard.svg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Salix alba

File:Simplified Control Flowgraphs.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Simplified_Control_Flowgraphs.jpg *License:* Public Domain *Contributors:* Joseph Poole, NIST

Image:SSA example1.1.png *Source:* http://en.wikipedia.org/w/index.php?title=File:SSA_example1.1.png *License:* Public Domain *Contributors:* Original uploader was Dcoetze at en.wikipedia

Image:SSA example1.2.png *Source:* http://en.wikipedia.org/w/index.php?title=File:SSA_example1.2.png *License:* Creative Commons Zero *Contributors:* Original uploader was Dcoetze at en.wikipedia

Image:SSA example1.3.png *Source:* http://en.wikipedia.org/w/index.php?title=File:SSA_example1.3.png *License:* Creative Commons Zero *Contributors:* Original uploader was Dcoetze at en.wikipedia

Image:Polytope model unskewed.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Polytope_model_unskewed.svg *License:* Public Domain *Contributors:* Traced by User:Stammered

Image:Polytope model skewed.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Polytope_model_skewed.svg *License:* Public Domain *Contributors:* Traced by User:Stammered

Image:Ogg vorbis libs and application dia.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ogg_vorbis_libs_and_application_dia.svg *License:* Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors:* Kővágó Zoltán (DirtY iCE)

Image:Ragel visualisation.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Ragel_visualisation.png *License:* GNU General Public License *Contributors:* Adrian Thurston. Original uploader was Hyperl at en.wikipedia

File:GOLD logo.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:GOLD_logo.gif *License:* Public Domain *Contributors:* Original uploader was DevinCook at en.wikipedia

File:GOLD Builder v3.4.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:GOLD_Builder_v3.4.gif *License:* Public Domain *Contributors:* Original uploader was DevinCook at en.wikipedia

Image:GOLD flow.png *Source:* http://en.wikipedia.org/w/index.php?title=File:GOLD_flow.png *License:* Public Domain *Contributors:* -

File:GOLD screenshot.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:GOLD_screenshot.gif *License:* Public Domain *Contributors:* Original uploader was DevinCook at en.wikipedia

File:Gccegg.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Gccegg.svg> *License:* GNU Free Documentation License *Contributors:* <http://gcc.gnu.org/img/gccegg.svg>

Image:purple_dragon_book_b.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Purple_dragon_book_b.jpg *License:* Fair Use *Contributors:* Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman

License

Creative Commons Attribution-Share Alike 3.0 Unported
<http://creativecommons.org/licenses/by-sa/3.0/>