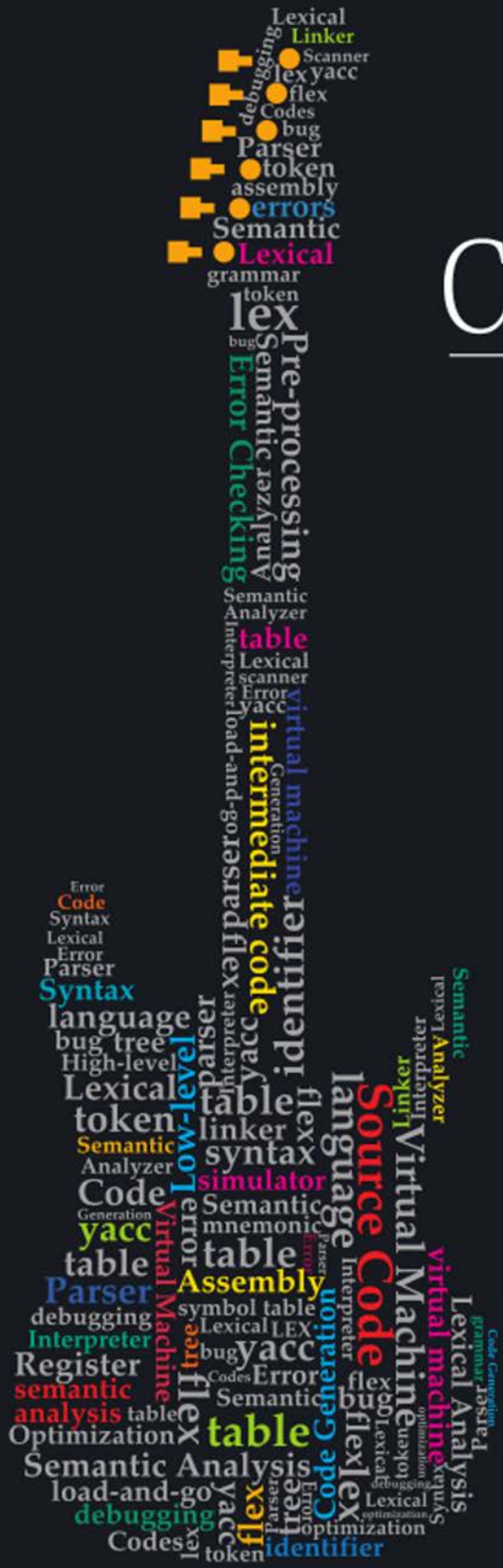

Lexical
Linker
Scanner
yacc
lex
flex
debugging
Codes
bug
Parser
token
assembly
errors
Semantic
Lexical
grammar
token
lex
bug
Pre-processing
Semantic Analyzer
Error Checking
Semantic
Analyzer
table
Lexical
scanner
Error
yacc
virtual machine
intermediate code
Generation
Interpreter
load-and-go
parser
identifier
Error
Code
Syntax
Lexical
Error
Parser
Syntax
language
bug tree
High-level
Lexical
token
Semantic
Analyzer
Code
Generation
yacc
table
Parser
debugging
Interpreter
Register
semantic
analysis
Optimization
Semantic Analysis
load-and-go
debugging
Codes
token
identifier
optimization
Lexical
Error
Semantic
Analyzer
Linker
Interpreter
Virtual Machine
Source Code
language
Interpreter
virtual machine
Lexical Analysis
Code Generation
grammar
parser
Syntax
optimization
Lexical
Error
Parser
table
flex
parser
Interpreter
Low-level
table
linker
syntax
simulator
Semantic
mnemonic
error
table
Assembly
symbol table
Lexical
LEX
yacc
Error
Semantic
table
flex
Code Generation
flex
Lexical
Error
Parser
Error
Lexical
optimization
Lexical
Error
Parser
yacc
flex
Interpreter
load-and-go
Codes
bug
flex
Error
Lexical
optimization
Lexical
Error
Parser
Lexical
Error
Parser
Lexical
Error
Parser

Compilers

Principles and Practice

Parag H. Dave

Senior Lecturer in Computer Engineering
Dharmsinh Desai University
Nadiad, Gujarat

Himanshu B. Dave

Ex-Professor and Head
Department of Computer Engineering
Dharmsinh Desai University
Nadiad, Gujarat

PEARSON

Delhi • Chennai • Chandigarh

To the countless unsung soldiers of the Indian armed forces, who selflessly gave up their today and tomorrow, simply so that we can have a today.

– Parag H. Dave

To Panini, Sanskrit Grammarian, who invented and used meta-language systems similar to Regular Expressions and BNF before Curent Era to precisely and compactly define the grammar of Sanskrit language.

– Himanshu B. Dave

Copyright © 2012 Dorling Kindersley (India) Pvt. Ltd
Licensees of Pearson Education in South Asia

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version. The publisher reserves the right to remove any material present in this eBook at any time.

ISBN 9788131764916
eISBN 9788131776117

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India
Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

Contents

List of Figures—ix
List of Tables—xv
List of Algorithms—xvii
Preface—xix
Acknowledgements—xxi

Chapter 1 Introduction 01

1.1 Languages—01
1.2 Translation Process—06
1.3 Translation Schemes—06
1.4 Theoretical Viewpoint—10
1.5 Phases of a Compiler—11
1.6 A More Detailed Look at Phases of a Compiler—13
1.7 A Real-life Compiler – `gcc`—19
1.8 What Do We Mean by "Meaning"?—20
Looking Forward—21
Historical Notes—22
Exercises—22
Web Resources—24
Glossary—24

Chapter 2 A Simple Translator 26

2.1 A Simple Language—26
2.2 Compiler for Simple—30
2.3 A Virtual Machine for Simple—44
Looking Forward—46
Exercises—47
Web Resources—47
Glossary—47

Chapter 3 Lexical Analyzer 50

3.1 Scanner—51
3.2 Symbol Tables and a Scanner—58

3.3 Compiler Writing Tools—61
3.4 Error Handling in a Scanner—70
Exercises—71
Web Resources—72
Glossary—73

Chapter 4 Syntax Analyzer 75

4.1 Top-down and Bottom-up Parsing—76
4.2 Top-down Parsing—78
4.3 Bottom-up Parsing—88
4.4 Yacc – A Parser Generator—113
4.5 Other Parser Generators—127
4.6 Grammar for `miniC`—128
4.7 Symbol Table and Parser—131
4.8 Real-life – GCC: GNU Compiler Collection—134
Exercises—135
Web Resources—137
Further reading—137
Glossary—137

Chapter 5 Syntax-directed Translation 139

5.1 Implicit Stacking in RDP—143
5.2 Synchronized Semantic Stacks—145
5.3 Action Symbols—152
5.4 Attribute Grammars—153
5.5 Symbol Table Handling—159
5.6 Intermediate Representation Output for `miniC`—161
Exercises—163
Web Resources—165
Further Reading—165
Glossary—165

Chapter 6 Type Checking 166

6.1 Data Types and Type Checking—167
6.2 Type Expressions and Type Constructors—168
6.3 Type Equivalence—171
6.4 Type Names, Declarations and Recursive Types—171
6.5 Type Inference—172
6.6 Type Conversion and Coercion—174
6.7 Overloading of Operators and Functions—175

6.8 Example: Type Checking in an Interpreter—175
Exercises—180
Web Resources—180
Further Reading—181
Glossary—181

Chapter 7 Run-Time Environment 182
7.1 Run-Time Storage Allocation—185
7.2 Operating System—216
7.3 Libraries—220
7.4 System Environmental Variables—222
7.5 Invocation Command-line Parameters—223
Exercises—224
Web Resources—225
Further Reading—225
Glossary—225

Chapter 8 Intermediate Code 227
8.1 Building a Parse Tree—229
8.2 Polish Notation—232
8.3 N-tuple Notation—237
8.4 Abstract Syntax Tree—239
8.5 Abstract or Virtual Machine Code—241
8.6 Threaded Code—244
8.7 SECD and WAM—248
8.8 Grammar and IR Generation for `miniC`—249
8.9 Real-life: Intermediate Codes of GNU `gcc`—260
Exercises—264
Further Reading—265
Glossary—265

Chapter 9 Code Generation and Machine-dependent Optimization 268
9.1 Our Concerns in Code Generation—269
9.2 The Target Language—275
9.3 Data Structures—278
9.4 Control Constructs—282
9.5 Procedures and Function Calls—283
9.6 The Target Operating Environment—287
9.7 Code Optimization—289

9.8 Machine-dependent Optimization—291
9.9 Converting the 4-tuple and RPN into Assembly Code—302
Exercises—307
Further Reading—308
Glossary—308

Chapter 10 Code Optimization 310

10.1 Basic Blocks—310
10.2 Value Numbering Scheme—316
10.3 Peep-hole Optimization—320
10.4 Structural Optimization—323
10.5 Global Data-flow Analysis—325
10.6 Super-optimizers—327
10.7 Epilogue—329
Exercises—330
Further Reading—331
Glossary—331

Chapter 11 Overview of Processing of Some Languages 332

11.1 Java—333
11.2 Perl—335
11.3 PROLOG—339
11.4 FORTH—341
Exercises—347
Web Resources—347

Chapter 12 Project: Compiler for a miniC 348

12.1 MiniC Language—348
12.2 Architecture of `miniC` Compiler—350
12.3 MiniC Grammar for `yacc`—351
12.4 Target Language—353
12.5 Symbol Table—361
12.6 Scanner—364
12.7 Parser—367
12.8 Code Generation—367
12.9 Testing—384
12.10 Use of `gdb` to Debug the FPU Operations—384
12.11 Difference Between AT&T and Intel Assembly Syntax—390

Exercises—391
Further Reading and Web Resources—391

Appendix A Formal Languages and Automata 393

A.1 Essential Mathematical Background—393
A.2 Formal Language Theory Review—410
A.3 Grammars—412
A.4 Regular Languages, Regular Expressions and Finite-state Machine—418
A.5 Context-free Languages, CFG and Push-down Automata—425
Exercises—431
Glossary—432

Appendix B Assemblers and Macro Processors 435

B.1 Assembly Languages—435
B.2 Assemblers—436
B.3 Macro Processors—445
Exercises—448

Appendix C Linkers and Loaders 449

C.1 Linkers—449
C.2 A Typical Linking Loader—452
C.3 In Linux World—456
C.4 Loaders—456

Appendix D Worked-out Problems 461

D.1 Problems for Chapter 4: Parsers—461
D.2 Problems for Chapter 5: Syntax-directed Translation—472
D.3 Problems for Chapter 6: Type Checking—475
D.4 Problems for Chapter 7: Run-Time Environment—476

Bibliography—481
Index—483

This page is intentionally left blank.

List of Figures

1.1 Genealogy of programming languages—06
1.2 What does a translator do?—06
1.3 A T-diagram—07
1.4 T-diagram of an assembler—07
1.5 A simplified view of an interpreter—08
1.6 Load-and-Go scheme—08
1.7 Perl and Python Load-and-Go—09
1.8 A simplified view of a compiler—09
1.9 A compiler in action—10
1.10 Compiler as an acceptor—11
1.11 Front- and back-end of a compiler—11
1.12 Multiple source, single target (m×1) and single source, multi-target (1×n) cases—12
1.13 Multi source, multi-target case (m×n)—12
1.14 Phases of a compiler and intermediate outputs—13
1.15 A finite state machine for example scanner—13
1.16 A syntax tree for example statement—18
1.17 `gcc` converts from C to obj using assembler—20
1.18 `gcc`: the Linker converts an obj file to an executable—20

2.1 A programmer's view of VM1—29
2.2 A token shifter used to supply next symbol—37
2.3 Linear two-address code array, called *matrix*—38
2.4 Typical display provided by VM1 simulator—45
2.5 Structure of VM1 simulator. Two threads are running – `cpumem` and `display`—45

3.1 Phases of a compiler: scanner—51
3.2 Two ways in which a scanner can interact with the parser, (a) Scanner creates an intermediate file of token-list; (b) Parser uses the scanner as a procedure and calls it when a token is needed—52

3.3 Compilation using translator writing tools `lex` and `yacc`. `Yacc` generates a full parser and makes provision for including *Actions* for semantic analysis and code generation, in form of program hooks. The dotted lines show this relationship—62
3.4 Building a compiler using `lex` and `yacc`. File naming conventions. For `flex` and `bison` these conventions are relaxed—63

4.1 Top-down and Bottom-up parsing—76
4.2 Starting from *I*, and looking at "*x*2", we derive a single tree, which ultimately results in "*x*2"—76
4.3 Starting from the given sentence, we detect the handle and reduce it by grammar rules—77
4.4 PDA consists of a FSM and a stack—79
4.5 Parsing action with PDA in a LL(1) type parser—82
4.6 Parsing function *M*(*X*,*a*) for simple LL(1) grammar—82
4.7 Parsing function *M*(*X*,*a*) for LL(1) grammar w/o ϵ rules—86
4.8 FSM for a parser for the example grammar—94
4.9 PDA: read/shift operation—95
4.10 PDA: reduce operation—95
4.11 A hierarchy of grammar classes—113
4.12 A compiler–compiler (theoretical)—113
4.13 YACC: options and outputs—114
4.14 Major components of `y.tab.c` file—115
4.15 Operation of `yyparse()`—116
4.16 A very simple grammar: file `example01.y`—116
4.17 A very simple grammar: file `example02.y`—117
4.18 Parser state diagram—119
4.19 A four-function calculator: file `calculator.y`—120
4.20 Overall structure of GCC – GNU compiler collection—135

5.1 Synchronized semantic stack used for semantic analysis. Note that the semantic stack does not push or pop every time the return stack does—145
5.2 Example of inherited attributes—156
5.3 Example of inherited attributes. One up-going arrow denotes a synthesized attribute, all remaining arrows indicate inherited attributes—157

7.1 Run-time environment of a translated program, apart from the basic environment that is the CPU—183
7.2 Growth of the three dimensions of the basic environment – processor facilities, memory sophistication and instruction set capabilities—184

7.3 Load-and-Go scheme. Left-hand side shows compiler based (e.g. Turbo-pascal) and right-hand side interpreter-based (e.g. Perl) schemes—185
7.4 Linker combines the object modules; loader assigns final addresses and loads them—187
7.5 Static allocation: activation record—187
7.6 Function call management in a legacy C compiler—189
7.7 A visualization of parameters layout with BP as the base—191
7.8 Process memory allocation for C—191
7.9 AR of the `main()` with alignment of the SP on 16-byte boundary—193
7.10 Activation record: no nested, no locals—195
7.11 Activation records pushed and popped—196
7.12 The block nesting structures for the example—197
7.13 Static and dynamic links in ARs—198
7.14 Heap: fixed sized blocks—203
7.15 Reference counting—205
7.16 A “block structured C” program—210
7.17 Stack symbol table operation—211
7.18 Preserving the discarded identifiers—212
7.19 Using an entry in the pointer array to represent a block—215

8.1 Intermediate language form of the program being translated links the analysis and the synthesis parts—228
8.2 Number of program modules required are $n + m$ for n languages and m machine architectures—229
8.3 A parse tree for input string `PLa = a * (a + a)`—230
8.4 A sample tree representation in memory—231
8.5 A parse tree for the example expression—239
8.6 A syntax tree for the example expression—239
8.7 An abstract syntax tree for the example expression—239
8.8 A typical stack machine—242
8.9 Subroutine threaded code – beginning of threaded code idea—245
8.10 Direct threaded code—245
8.11 Indirect threaded code—246
8.12 GNU compiler collection framework. Generic, gimple and RTL are the Intermediate codes in GCC—261

9.1 Code generator in the compiler chain—269
9.2 Three “dimensions” of orthogonality of i nstruction set—271
9.3 A vector of integers and three versions of access to its elements—279

9.4 Layout of an array in the memory—280
9.5 Call parameters' access—284
9.6 A standard x86 stack frame—284
9.7 Creation of an AR during a function call—288
9.8 A visual showing positions of parameters in an AR—288
9.9 Memory hierarchy—290
9.10 Instruction execution pipeline—301

10.1 An example flow-graph—313
10.2 Peep-hole optimization—320

12.1 Processing in miniC compiler—350
12.2 Front-end of miniC compiler—350
12.3 A part of the DFA model of the scanner—364

A.1 Mathematics used for modelling—394
A.2 A member of a set type can be INT, STRING, SET, LIST, EMPTY, DUMMY. Note the union—400
A.3 A set consists of a set header and a singly linked list—400
A.4 An acceptor – if it says 'Yes' the string is in the Language, if 'No' then it is not—413
A.5 Relationship between a language, grammar and acceptor—413
A.6 Railroad diagrams: for top – a statement, a simple expression using only addition of terms, a term and a factor in an arithmetic expression. Compare with corresponding grammar—415
A.7 An abstract syntax tree for example grammar—416
A.8 State diagram of an FSM M_2—420
A.9 A NDFSM. Note the multiple transitions for '0' at state q_1 and "spontaneous" transition denoted by ϵ—421
A.10 An example NDFSM, M_4—422
A.11 String s made up of u, $v^m w$ related to the transition diagram—424
A.12 A FSM recognizing language L_3—424
A.13 Two possible *derivation* trees for $a * a - a$—427
A.14 A push-down automaton. FSM part in heavy lines—428
A.15 A derivation for string "$a * (a + a)$"—430
A.16 The three "jig-saw" pieces which make up string `a * (a + a)`—430
A.17 Some of the combinations of the three jig-saw pieces; first generates string `a * a`, second `a * (a + a)` and third `a * ((a + a) + a)`—430
A.18 The five substrings involved in pumping lemma—431

B.1 A small assembly language program—436
B.2 T-diagram of a cross-assembler—437
B.3 A symbol table entry using separate name storage—440
B.4 Hash table: basic idea—441
B.5 Hash table: collision resolution—441

C.1 Linking process—450
C.2 Several modules to be linked—450
C.3 Minimum information which needed to be present in an object module—451
C.4 Logical view of process memory—452
C.5 Actual process memory. Note the “free” area—452
C.6 Static linking—453
C.7 Dynamic linking—453
C.8 Memory layout for ELF—456
C.9 Page frames in the physical memory are assigned as pages to processes. All pages of a process need not be in the physical memory simultaneously—459
C.10 Virtual memory management—460

D.1 Call graph and call tree—477
D.2 Call graph – note that there is no cycle—479
D.3 AR showing local variables—479

This page is intentionally left blank.

List of Tables

2.1 Some of the OP-codes of VM1—29
2.2 Mapping from intermediate code to VM1 code—43

3.1 Parsing: scanner and parser—50
3.2 Token value component for different atoms—52
3.3 Pattern-matching primitives—64
3.4 Lex: pattern matching examples—65
3.5 Lex pre-defined variables and functions—66

4.1 Parsing: scanner and parser—75
4.2 Left and right derivations, right reduction—78
4.3 Simple LL(1) parser: input string `b+b*a#`—83
4.4 Simple LL(1) parser: input string `adbbeccbce#`—84
4.5 Parsing of example sentence `i[e]=e#`—88
4.6 Precedence relations for the example grammar—91
4.7 LR(0) parser actions for the string `a*(a+a)#`—96
4.8 Functions F( ) and G( ) for example grammar—101

7.1 Some of frequently used linux system calls—219

8.1 Examples of RPN generation – simple statements—234
8.2 Examples of RPN generation – IF constructs—236
8.3 Examples of RPN generation – WHILE construct—237
8.4 Examples of AST generation—240

10.1 Strength reduction transformations—321

12.1 Data transfer instructions—354
12.2 Integer instructions—355
12.3 Logic instructions—356
12.4 Flow control instructions—357
12.5 Assembler directives—358
12.6 Important linux system calls—383

A.1 Examples: assertions—394
A.2 Classification of languages as per Noam Chomsky—416
A.3 Recursive definition of a regular expression and regular language—418
A.4 Relation between FSM and basic operations in RE—421
A.5 Transition function of an example PDA—429
A.6 Operation of example PDA—429
A.7 Simple meanings of some common terms—432

C.1 Example: modules to be linked—451

D.1 Parsing function for grammar in problem 3—462
D.2 Parsing function for grammar in problem 4—463
D.3 Parsing function for problem 5—463
D.4 Parsing table for grammar in example D.1.7—464
D.5 Parsing of an example string `wyyz#`—467
D.6 Parsing functions *f*( ) and *g*( ) for problem 12—469
D.7 Trace of parse of `a + a ^ (a + a) / -` problem 12—470

List of Algorithms

2.2.1 lookup(s)—33
2.2.2 install(s, t, ad)—33
2.2.3 Scanner—33
2.2.4 pE( )—36
2.2.5 pplus( )—36
2.2.6 pLET( ) function—39
2.2.7 pE( ) – returns a token—39
2.2.8 pF( ) – returns a token—40

4.3.1 Operator precedence parsing—92
4.3.2 Table-driven LR(0) and SLR parser—100

8.2.1 Evaluation of RPN—233

10.1.1 Basic blocks—311
10.1.2 Next reference—313
10.2.1 ValueNumbering(BB)—318
10.5.1 Iterative Live Analysis—327

B.2.1 Pass-I—438
B.2.2 Pass-II—439
B.2.3 Macro-processing Pass-I—444
B.2.4 Macro-processing Pass-II—445

C.2.1 Linker Pass-I—454
C.2.2 Linker Pass-II—455
C.4.1 Absolute loader—457

This page is intentionally left blank.

Preface

"Why learn compilers?" – that is a question many of our students have asked us in our teaching career. We have given various answers – sometimes serious, sometimes only partially serious, but here are two quotes which we like:

> *"If you don't understand compilers, you can still write programs – you can even be a competent programmer – but you can't be a master."*
>
> – Hal Abelson, MIT

> *"If you don't know how compilers work, then you don't know how computers work. If you're not 100% sure whether you know how compilers work, then you don't know how they work."*
>
> – Steve Yegge

The study of compilation techniques is integration of many things a Computer Science/Engineering student learns separately – Data Structures, Algorithms, Theory of Automata and Formal Languages, machine architecture, even operating systems. The only other CS course which integrates to a comparable extent is a course on Operating Systems, but there the emphasis is different.

There are several good books on Compilers available to students. Some are highly theoretical and give limited details of the practical aspects of compiler writing. Some of them are filled with detailed code of a compiler. They are very difficult to be used as a textbook in an undergraduate engineering course.

The available textbooks represent a spectrum of presentation approach. Some of the books are voluminious with considerable amount of theoretical material that are more suitable for those students who have "Compilers" as a research interest. At the other end of the spectrum are books that contain pages after pages of C code of not only compiler proper, but compiler-writing tools also, with just sufficient theory. Our book is somewhere between these extremes in this spectrum and we feel that our book is more suitable for students at Indian universities.

We have tried to strike a balance between Theory and Practice. The full source code of a compiler is available at www.pearsoned.co.in/paraghimanshudave.

The contents and style of presentation are especially suitable to Indian students. To enhance the students learning experience, we have given a Glossary, Suggestions for Further Reading and Web Resources at the end of almost all chapters.

A distinguishing feature of the book is a detailed introduction to compiler writing tools and GNU `gcc` (GNU Compiler Collection), which we believe, provide an excellent platform for those students who want to take up research in compilation and optimization techniques and engineers who want to implement compiler for new languages.

The chapters are sequenced like this: After *Introduction* in Chapter 1, we discuss a complete compiler scheme for a Simple language in Chapter 2 – *A Simple Translator.* We then start our

discussion of various phases of a compiler in Chapter 3, where we discuss *Lexical Analyzers.* In Chapter 4 we explain *Syntax Analyzer* at length. Next *Syntax Directed Translation* is elucilated in Chapter 5. *Type Checking*, which has invoked considerable theoretical interest is described in Chapter 6. Chapter 7 details *Run-time Environment. Intermediate Code* issues are examined in Chapter 8. In Chapter 9 we have discussed *Code Generation and Machine Dependent Optimization,* where we have also given considerable details about programming of a Floating-point processor for a specific popular architecture (x86). Chapter 10 deals with *Code Optimization* issues. We thought it worthwhile to take an *Overview of Processing of Some Languages* to Chapter 11, where we have considered Java, Perl, PROLOG, and FORTH from the processing viewpoint. Finally, in Chapter 12 we give details of a *Project: Compiler for a miniC* to conclude the study of compilers.

Three Appendices – *Formal Languages and Automata, Assemblers and MACRO Processors* and *Linkers and Loaders* provide the essential background material. The fourth appendix *Worked-out Problems* gives several problems with solutions.

A Bibliography and an Index given at the end should help locating reading material.

There is enough material for a 40 to 45 1-hour lecture course at the undergraduate level.

Acknowledgements

We used computers running on Linux O/S for preparing the Draft of this book, including all the drawings, demonstration program code, text processing, spelling check, index preparation, etc.

The authors thank Linux and GNU community – starting with Linus Torvalds and Richard Stallman – for providing a dependable, elegant and pleasant working platform. We especially appreciate the fact that while we were preparing this book, we did not have to worry about viruses, bloated system resources usage or the "Blue Death".

We used Latex (Leslie Lamport) and Tex (Prof. Donald Knuth) for the text preparation. `XFIG`, `PSTricks`, `gnuplot` and `metapost` were used to prepare various diagrams, graphs etc. Perl (Larry Wall) was used extensively for all onetime jobs and also a few compiler routines. `Ispell` and GNU Aspell were used for spell-checks. We have used the "Higher Order calculator" (HOC) given in Kernighan and Ritchie book as a model on which our `miniC` example language is based. The Index was prepared using our own semi-automatic index generator which was created as a part of academic work of one of us.

The text in this book will show that not only did we use these software, but we learned from them also. We heartily thank known and unknown authors of these software.

Himanshu B. Dave: At this moment I would like to remember all my teachers right from school onwards. Though I was not fortunate enough to study at a top-class school, I did have top-class school teachers – Shri Krishna Balavant Tarkunde taught mathematics. Learning geometry and calculus from him was a pleasure. He continued to be my guide even during my college days. Shri Fulchand V. Shah taught me drawing and painting and it was he who detected my liking for science and told me to pursue science and technology rather than arts. Shri A. C. N. Desai taught English and Shri A. K. Choksi taught Sanskrit. Shri Topiwala, my science teacher, allowed me to tinker in the physics laboratory, which helped later in my engineering studies. Shri Poonawala taught algebra and also English grammar when he got into the mood.

There is one non-traditional teacher during my school days, whom I remember fondly – Shri Shinde. He had a radio repair shop in Surat and I joined him as a part-time assistant to learn radio technology. He was the first to teach me real-life debugging techniques, which later proved to be an invaluable asset.

During my two years at the M. T. B. Science college, I was most impressed by Prof. Pradyumna Anandji Pandya who taught us Physics and Prof. Jayadev Atmaram Kazi, who taught Chemistry. The "Gujarati Poetry Appreciation" classes by Prof. Jayant Pathak were most enjoyable.

In the Engineering college at Ahmedabad, I had some good teachers – Prof. Anant Keshav Kulkarni taught me Electronics and allowed me to play in the Electronics laboratory. The experiments and equipment in U. H. F., which I built under his guidance, were instrumental to some extent for my entry into IIT Kharagpur. Prof. Bengali taught Electrical Machines and made us work out difficult problems. Prof. Kalwachwala, who later became the Director of Technical Education, showed that an engineer should not run away from a difficult situation, when there was an incidence of fire in our Electrical laboratory.

During my long stay at IIT Kharagpur – initially for studies and later as a teacher myself – I had a good fortune to be taught by and associated with Prof. G. S. Sanyal, Dr J. Das, Dr B. Das, Prof. Gagan Bandyopadhyaya, Dr M. K. Achyutan, Dr Bose, and Dr Hrishikesh Rakshit. Prof. Amarendra Pandya and Shrimati Jyotsna Pandya provided a home away from home.

After my retirement, I took up a job as Director (Technology and HR) at a private Internet Service Provider for a short period. That was a sudden jump from academics to real-world and I learned quite a bit during those two years.

At present, I am associated with a high-tech IT Services company *elnfochips,* as Senior Consultant (Training). I would like to thank Shri Pratul Shroff, Upendra Patel, Sudhir Nayak, Prajose John, and Ms. Aanal Patel, for providing a comfortable and free environment and extending their love and respect. There, I have taught young, fresh engineers to scale the wall separating the academia and real-world industry, and in the process, also learned a lot myself.

In the end I want to remember my parents – Shri Bhalchandra K. Dave and Shrimati Sarayuda B. Dave. My father always supported me in various ways in my tinkering with electrical gadgets. He would buy tools and components for me as per my growing requests. He also further taught me Sanskrit and instilled in me interest in our scriptures – Upanishads and Vedas, a fact which I appreciate very much. He, along with my mother, introduced us children to Indian Classical music and encouraged us to pursue it as a hobby.

When one looks back at all the past history, one realizes that human race is a continuum – from father to the son, from the teacher to the student. So, in the end, here is thanks to all the teachers, who represent the ultimate Teacher-par-excellence.

1
Introduction

What you will learn in this chapter

- Which languages are we talking about?
- Translation process
- Various schemes of Translation
- Types of Compilers
- Why are High Level languages so heavily used?
- What does a compiler actually do?
- Phases of a compiler
- Theoretical viewpoint regarding a compiler

Key Words

High Level language, bytecode, translation, compilers, phases, interpreters

1.1 Languages

If you think a little about how our world operates, you will realize that animate and inanimate objects interact by sending messages to one another. These messages take various forms. For example, a honeybee communicates the location of a source of juices to other honeybees by a kind of "dance". We human beings "talk" with one another or send written or electronic messages. You "talk" *to* (as distinct from *via*) your mobile phone by pressing various buttons and reading the display. In that case, your pressing the phone buttons is called *commands* or *requests*, but still, in a general sense they are *messages*.

All messages are expressed in some language, and usually there is a protocol (rules of communicating) for any situation where such messages are exchanged. For example, when someone calls Siddhartha on his phone, protocol requires that he says, "Hello, Siddhartha speaking".

Various races of human beings have developed a number of languages for communication over thousands of years, such as Sanskrit, Farasi, Chinese, English, French, Tamil, Telugu, Hindi, Eskimo, Swahili, etc., in fact a few thousands of them. These are all called *Natural languages*, as they were developed during the natural evolution of human races.

When the era of computers and computer-like devices started during 1940s, a need arose for communicating with the computing machines. Most of the communication was in the form of commands or requests to do some work or job. As the machines "understood" only very elementary

commands ("add number x to number y and put the result in a store called z") rather than jobs of even mild complexity ("solve the quadratic equation $ax^2 + bx + c$, given a, b and c"), it was necessary to supply a complete sequence of elementary commands, specifying how a particular job was to be done. A sequence of commands is called a *program*. This necessitated developing and defining the so-called *programming languages*.

During the early days of computer evolution, the only language a computing machine could follow was its binary command language (as most, though not all, machines were built using binary logic elements – called gates and flip-flops). This binary language was extremely terse, difficult to use and error-prone. With development of technology, it was found that programming can be done in languages more approachable for humans, but this entailed a price. If we call the language understood directly by a machine as M and the language more desirable from human viewpoint for specification of a computing job as L, then there should be a method available to *translate* a code in L to a code in M. Who would do this work? In fact, in the old days of the computer era, we understand that in some countries like Russia, this work was done by human clerks, called (aptly) "translators". In the computer field, there are several interesting adages, one of them is: *Let the computer do the dirty work*. So, why not make the computer do the translation from L to M? The question is how can a computer machine, which does not "understand" any language other than its own command or machine language, translate from L (a language it does not understand) to M (a language it understands)? If you do not know Chinese, but know English, can you translate from Chinese to English?

This seemingly impossible work is made possible by using certain properties of the programming languages. Unlike the natural languages, the programming languages have a very restricted structure. Almost "blind" (i.e. mechanical) replacement for various words or constructs in a human approachable programming language like L, by a corresponding construct in M, is possible. This mechanical work can be performed by a computing machine.

This is a very brief account of translation process. For slightly more details read the rest of this chapter, and a still fuller account is given in this whole book.

A systematic study of both programming and natural languages requires that mathematical or *formal* models of various types of languages be defined. The languages which define such models are called *Formal languages*. In order to capture the essence of a language and allow mathematical manipulations, these formal languages look somewhat different from our day-to-day sense of a language. Appendix A on "Theory of Automata and Formal Languages" gives a detailed discussion of this topic.

Some years back, pioneering linguistics researcher, Noam Chomsky, set upon the task of translating a document written in one natural language, say Japanese, into a corresponding document in another, say English. While trying to develop the algorithms to do this difficult job, he studied the nature of various types of languages. A summary of his main findings are given in Appendix A, as Chomsky's Hierarchy of languages.

In this book, we concentrate only on translation of programming languages, taking models from formal languages where required.

A computer professional may come across several types of programming languages. The following sections details majority of them.

1.1.1 Machine Language

The binary machine code is one which the computer hardware "understands" is able to interpret for execution. A machine language program consists of *instructions* in binary digits, i.e. *bits* 0 and 1.

The machine instructions of modern computers consist of one or more *bytes* – group of eight bits. For example, some typical binary instructions for 8086 CPU chip are:

```
binary code         Mnemonic opcode    instr length (bytes)
11111000              clc                 1
00000000 11000001     add al, cl          2
```

Obviously, if one has to program in such a language, it is going to be very tedious, time-consuming and error-prone. Another difficulty would be while loading such a binary program from our paper to the memory of the computer.

In the old days of computer technology, the initial boot program (IPL), which loads a operating system from some auxiliary memory to the main memory, was required to be keyed-in via console (binary) key switches. The operator would have to remember the binary code for the few instructions of the IPL and painstakingly key them in.

1.1.2 HexAbsoluteLoaderLanguage

The above difficulty was overcome to some extent, by writing a Hex loader program in machine language, which could read ASCII representation of the machine instructions in hexadecimal. For example, the above two instructions would look like: F8 and 00 C1, respectively. Generally, such a loader program accepted program to be loaded in the format:

```
<load address> <length> <bytes to be loaded> <check-sum>
   000C200        50     4F34 ...   ...   ...  87E3
```

This was still a far cry from a human-friendly programming language.

1.1.3 Assembly Language

The earliest attempt at providing a somewhat human approachable programming environment was in the form of *assembly language* and its translator, called an *assembler*. An assembly language provided:

- *mnemonic* op-codes – the programmer need not remember binary or hex op-codes;
- *symbolic* operands – the programmer need not keep track of the absolute memory addresses where various quantities in his program resides;
- *address arithmetic* – programmer can express addresses as displacements with reference to some declared symbols;
- *data declaration* – predefined data (initialized data) can be declared;
- *memory reservation* – working memory areas can be reserved and named.

For many years, even major applications were written in assembly language, extended by the macro assembly language (Section 1.1.4). Even today it is used as a step in translating High Level languages and also where hardware-specific code is needed, for example, in operating systems. Because each family of CPU has different instruction set, there is an assembly language (and a corresponding assembler) for each such CPU family. Thus, an assembly language reflects the architecture and *Instruction Set* of the computer for which it is designed.

1.1.4 Macro Assembly Language

With some experience of programming in an assembly language, programmers found that most of the time they are using repetition of basically same or similar code sequences. For example, summing

up an array of numbers may be required as a commonly used operation. In a macro language, such sequences can be defined as a *macro* and be given an identifier. Once this is done, wherever in the remaining code one wants to insert the code, only the macro name need be specified.

Macro assembly language largely removed drudgery from programming. IBM "Autocoder" was once a much used such macro language.

1.1.5 Intermediate or ByteCode

ByteCode represents the "machine language" of a phantom or *virtual* computing machine. It is used as an intermediate representation of a program, keeping only the most essential features of the program. For example, compiler translates your `myprog.java` source code into `myprog.class` file which contain bytecodes.

1.1.6 High Level Language

A High Level language (HLL) looks more like a natural language than a machine or assembly language. An HLL is characterized by the following:

- It consists of statements, which specify steps of computation, in contrast to machine instructions in assembly languages.
- Statements are made up of language atoms – numbers, characters, text strings, function calls, etc.
- Usually, four types of statements are available:

  Declarative: declares existence of certain entities having specified properties;

  Definition: gives definitions of user-defined entities;

  Imperative: specifies commands to be passed on to the execution agent (usually a CPU) – one very common imperative being an assignment;

  Assignment: specifies some computation to be done and assignment of resulting value to an entity;

  Control: determines the order in which statements are executed, generally by checking the current state of some entities;
- Structured programming and modularity are generally emphasized.
- Libraries of precompiled and tested modules are available for computation over diverse fields.
- Designed for easy debugging, code modification and code extension.

C, C++, Java and Ada are examples of HLL.

1.1.7 Very High Level Language

As advantages of HLL languages were realized (see Section 1.1.8), very High Level language (VHLL) was developed, especially for some specific application areas or computational needs. Apart from carrying forward the facilities generally available in a typical HLL, additional language facilities are seen in such languages. For example,

- PROLOG is a language for computation based on formal logic and specifically *Horn clause*. It is a declarative language for logic programming as opposed to procedural languages such as C or Java. It declares the data elements and relationships between them. Its basic action is to automatically search for a combination of values of specified entities to satisfy some logic expression. It does not depend upon step-by-step algorithms as we normally encounter.

- The programming languages for simulation package like MATLAB or statistical package like **R** are VHLL. They are object-oriented languages.
- Haskell is a *Functional* programming language.
- Perl and Python can also be called VHLL, as they use higher level objects such as lists, array and dictionaries directly. For example, a simple statement in Perl,

```
@evens = map {$_ if $_ % 2 == 0} (0...100);
```

  will assign a list consisting of all the even values between 0 and 100 to array @even. The iteration over an array of integers 0 to 100 is handled automatically as a part of the `map` function.

- The METAFONT language developed by Prof. Donald Knuth, as a part of his TEXsystem to design fonts, is a VHLL. It automatically solves simultaneous equations to obtain co-ordinate values from known points.

1.1.8 Why High Level Language?

- **Readability:** An HLL will allow programs to be written that look similar to a text description of the underlying algorithms. If some care is taken, the program can become *self-documenting.*
- **Portability:** Though an ideal only partially achieved, HLL, being mostly machine independent, can be used for developing portable software. The same source code can be used on different families of machines, only thing needed is that the code has to be compiled on each type of machine separately.
- **Productivity:** Due to the brevity of expression in HLL for a given algorithm, the programmer's productivity increases. A rule of thumb used in software industry is that a programmer is expected to deliver 50 lines/day of tested and debugged code in any implementation language. Processing specified by 50 lines of code in C will be much higher compared to 50 lines of assembly code. Naturally, the higher the level of the language of the code, higher will be the programmer's productivity.
- **Debugging ease:** As the statements implementing the algorithms have logical meaning, finding and removing errors (debugging) are much more easier. Structured control constructs, if used in a proper way, also helps in debugging. Some HLL provide object-oriented programming facilities which further enhances debugging ease, modularity and code reuse.
- **Modularity:** The complete application development work can be divided into modules. This aids team work and also leads to *code reuse*, which saves considerable effort in code development.
- **Optimization:** Most of the good compilers provide, as option, optimized code. This executable code is comparable to what a human expert would deliver if he is using an assembly language.
- **Generality:** An HLL generally allows writing of code for a wide variety of applications, thus obviating a need for the programmer of becoming expert in many different languages.
- **Error detection:** An HLL compiler can detect several types of programmer errors, e.g. using a real variable as an integer, or not passing enough number of call arguments to a function. Also, a good compiler can give warnings for doubtful but syntactically correct code. Such warnings many times lead to detection of *logical errors* in the code, which a compiler is normally not expected to detect.

The HLL was developed during the 1950s to till today. Many of the HLLs are related to some others, and this relationship is called the genealogy of the programming languages (see Fig. 1.1).

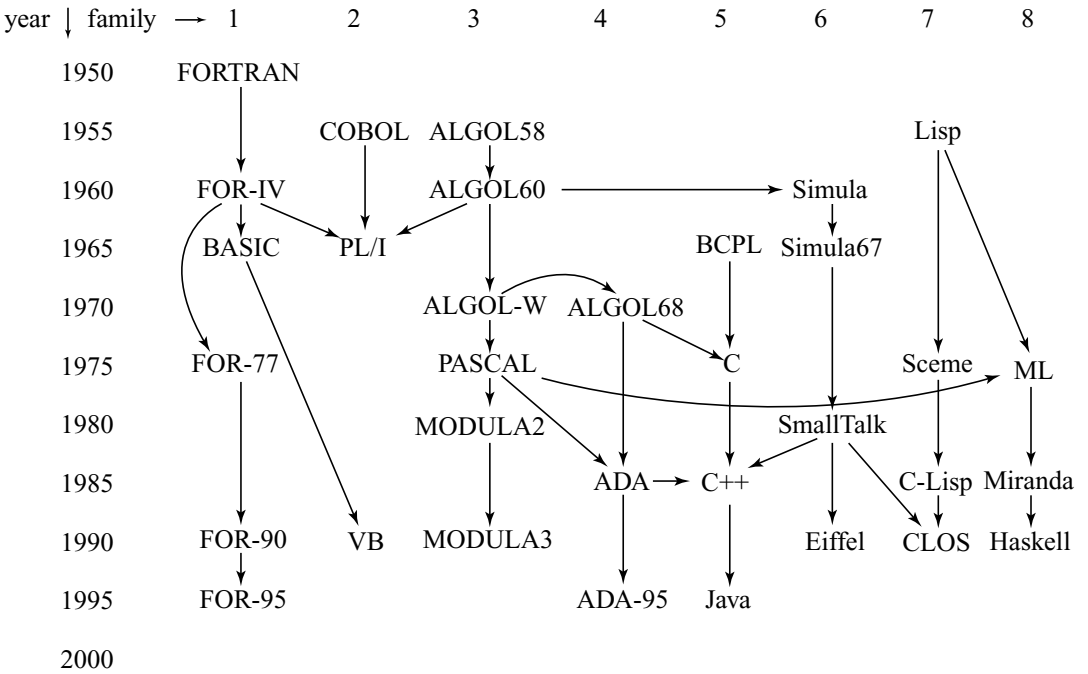


Fig. 1.1 Genealogy of programming languages

Note: The vertical axis in years is approximate only. Even though a language may be related or derived from some others, it does not mean that their run-time environments are comparable.

1.2 Translation Process

A programming language translator takes a source code written in one language and generates an output code in the target language, for later execution on some execution agent. Sometimes, the output is further processed by another translator to a second target language (see Fig. 1.2).

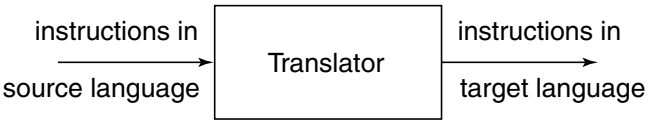


Fig. 1.2 What does a translator do?

Note: One important requirement – the algorithm specified by the source code should remain intact across the translation, i.e. the target code must specify exactly the same algorithm.

1.3 Translation Schemes

1.3.1 T-diagram

Most of the translation schemes accept some input code in a specified source language, translates it using a program which is written in its implementation language and generates an output in a specified

target language, as shown in Fig. 1.2. The translation program itself works on some specified system – a combination of hardware and operating system software. To depict this arrangement in a compact way, the so-called T-diagrams are traditionally used. A general T-diagram is shown in Fig. 1.3. The arrow over the middle upper box indicates the direction of information processing. Sometimes, we may have to extend this symbolism by adding more components such as, e.g., the hardware platform on which the translator will work. In that case one more box will be added below the middle lower box.

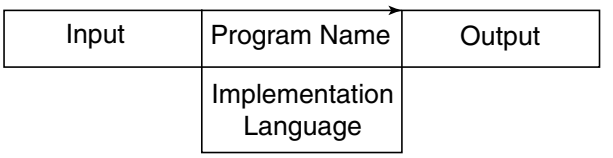


Fig. 1.3 A T-diagram

1.3.2 Assembler

An assembler translates an assembly language source code to executable or almost executable (object) code (see Fig. 1.4).

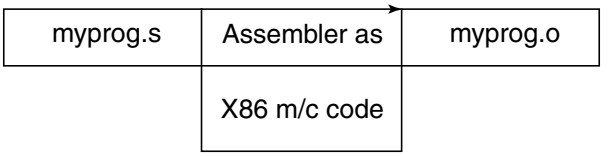


Fig. 1.4 T-diagram of an assembler

Basic operation in an assembler is the replacement of symbolic information by numeric (binary or hex) information. It does this by help of two tables – one fixed – op-code table – and the other built-up when the source is read and analyzed – symbol table. The assembler will create and use a *Symbol Table* to keep track of the identifiers used in the source code.

An assembler is generally used as one of the steps in a typical modern compiler. For further details, see Appendix B, “Assemblers and Macro processors”.

1.3.3 Macro Assembler

Though it is also loosely called *Macro processor*, there is some difference in meaning. A macro processor is any program which accepts macro definitions in some macro language and provides for macro *expansions* or substitutions. It could be a stand-alone system like `M4` macro processor used heavily in UNIX/Linux systems, or it could be a part, as an additional layer of facility, of other software – text editors, spreadsheets, compiler, etc.

On the other hand, macro assembler is a macro processor integrated into an assembler. It is an assembler with built-in macro facility. For further details, see Appendix B, “Assemblers and Macro Processors”.

1.3.4 Interpreter

An *interpreter* accepts a source code as input and immediately executes it. It performs analysis of each statement in the source code to find its *meaning* and performs those specified operations, using the operating system and hardware platform on which it is based (Fig. 1.5).

Fig. 1.5 A simplified view of an interpreter

There is statement by statement processing of the source code, but there is no one-time translation of the whole source code. Each time a statement needs execution, it has to be processed afresh. This takes time, of course, and shows up as slower execution compared with a compiled program. How much slower? It could be as slow as 5 to 100 times, depending upon the source language and implementation details.

For example, consider a small script segment in shell-script – an interpreted, *scripting* language for the *shell* of UNIX/Linux systems:

```
for f in *.c
do
  grep -n "function" $f
done
```

This script segment considers all files in current directory, having file name extension '. `c`', one by one, and prints all the lines in each of them which have the word "function" in it. Every time the `for` statement is executed, it is analyzed afresh to find its "meaning", similarly for the `grep` statement etc.

1.3.5 Load-and-Go Scheme

A substantial use of an HLL is in "one time" programs which are required to be written and checked out quickly as there is an immediate but short-term need for that program. Also, students' programs are for learning the HLL language and programming techniques rather than long-term or commercial use. In such situations, a *Load-and-Go* scheme will be desirable. In such an arrangement, the text editor via which a program is developed and the compiler (or interpreter) are integrated in one package. The user types in his/her program using the built-in editor and the program is immediately translated and executed. A considerable simplification in the compiler and Linker takes place as the user program will be always loaded at a fixed location in the memory. The required function library is also in a fixed place, which makes the linking operation easy (see Fig. 1.6).

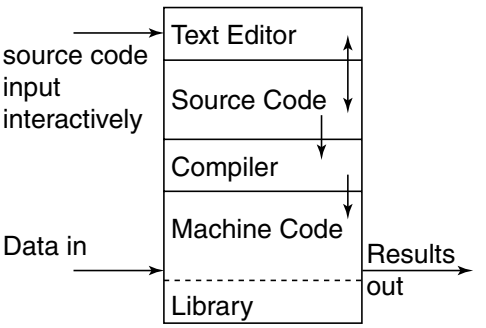


Fig. 1.6 Load-and-Go scheme

Advantages: Simpler compiler and linker, fast turn-around on experimental programs, easier to debug.

Disadvantages: Useful only for small programs, usually limited to a single source file. Library extension is almost impossible. Code optimization is not present due to need for keeping the compiler simple and small.

Similar scheme is implemented for Perl and Python script processors, as shown in Fig. 1.7.

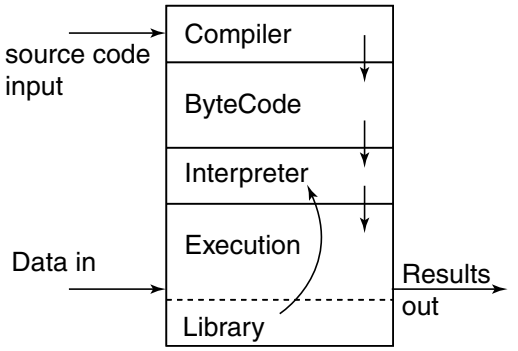


Fig. 1.7 Perl and Python Load-and-Go

1.3.6 Compiler

A compiler takes a source code in an HLL as input and generates either a machine code executable or an object code for subsequent linking and execution (Fig. 1.8).

Fig. 1.8 A simplified view of a compiler

There are several types of compilers.

One-pass compilers: The compiler completes all its processing while scanning the source code only once. It has advantage of simpler and faster compiler, but it cannot do some of the sophisticated optimization. In the old days, the main memories were relatively small (a few hundred kilobytes of memory was considered a luxury) and the intermediate outputs during translation were stored on magnetic tapes. In modern times, high-speed hard-disk drives and much larger memories are norms, and distinction between one- and multi-pass compilers is dimming.

Multi-pass compilers: The compiler scans the source code several times to complete the translation. This allows for much better optimization. It also takes care of some quirks of the HLL being handled. For example, consider the following three statements in FORTRAN:

```
DO 11 I = 1, 10
DO11I = 1,10
DO 11 I = 1. 10
```

The first and the second statements mean exactly the same thing, because in FORTRAN spaces are simply ignored, they do not have any role in separating out atoms of the language. Now look at the third statement, it has only one character different from the first two, but the meaning is very much different. While the first and second statements were header of a DO-loop, the third is an assignment statement. Thus after the FORTRAN compiler detects a ‘,’ between ‘1’ and ‘10’, it will have to go back and re-scan from beginning to separate out the atoms ‘DO’ ‘11’ I ‘=’, which it

need not do if the character between '1' and '10' were '.'. In that case, the atoms would be 'DO11I (a valid identifier)' '=' '1.10'.

Load-and-Go compiler: We have already discussed this in Section. 1.3.5.

Optimizing compilers: They contain provisions for target code optimization, so that it is efficient in terms of execution speed and memory usage. Almost all modern compilers do have such facility, generally as a number of options.

Just-in-time compiler: Used by Java and by Microsoft .Net's Common Intermediate Language (CIL). Here, an application is supplied as bytecode, which is subsequently compiled to machine code of the platform *just prior to execution*.

1.3.7 What Does a Compiler Do?

To summarize what we have discussed till now about a compiler, we note that a compiler (see Fig. 1.9):

- Translates a user program in one language L1 into a program in another language L2.
- It is a large set of programs, with several modules.
- L1 (source) is usually a High Level language like C, C++ or Java.
- L2 (target) is usually a form of the binary machine language.
- L2 is not a pure machine language, because two further operations *Linking and Loading* are needed before the program is in executable form. For a detailed discussion of Linking and Loading operations, see Appendix C "Linkers and Loaders".
- It consists of several steps or *phases*.

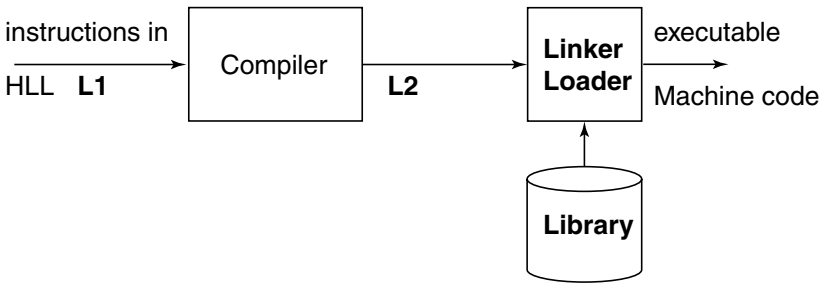


Fig. 1.9 A compiler in action

1.4 Theoretical Viewpoint

Before we go ahead with knowing more details about a compiler, it is worthwhile to know what the formal language and automata theory have to say about a compiler. A more substantial account is given in Appendix A, but here we have bare minimum.

1.4.1 Acceptor and Compiler

In formal language theory, a compiler is simply an acceptor, which reads input strings and outputs "Yes" or "No", depending upon whether a string is in language L1 or not (see Fig. 1.10).

A compiler is useless if it does not point out errors. (Yes, but which errors?) A real compiler does this by typing out errors! It also translates from L1 to L2, but the *acceptance* function is still the most

important. What will you do with a program which is not correct according to the syntax of the source language?

It tells you whether your program adheres STRICTLY to the rules of a particular language. These rules are given as a *grammar* and a compiler represents this grammar.

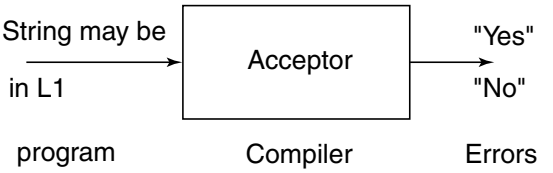


Fig. 1.10 Compiler as an acceptor

1.5 Phases of a Compiler

Most of the modern compilers have the following phases:

- **Pre-processing:** Usually, this phase is implemented as a macro processor, with its own macro language. It provides for file inclusion, conditional compilation control of segments of the source code, definition of literals, "pragma" commands to the compiler, etc.
- **Lexical analysis:** (Scanner) – checks for correct use of the input character set, identifies language atoms and tags them as the type of atom, e.g. NUMBER, IDENTIFIER, OPERATION.
- **Syntax analysis:** (Parser) – processes output of the Scanner, detects syntactic constructs and types of statements, generates a *parse tree*.
- **Semantic analysis:** (Mapper) – processes the parse trees, detects the "meaning" of statements and generates an intermediate code.
- **Code Generation:** Processes the intermediate code to generate the "machine code" for the target machine. Or alternately, works as pseudo-code interpreter to execute the intermediate code directly (e.g. Perl, Java, Python). In that case it may be called a *Virtual Machine* (VM).
- **Error checking:** (Spread throughout the compiler) – various kinds of error checking, appropriate for a particular phase, are done in each phase.
- **Optimization:** (Spread among several phases) – two kinds of optimizations are done – machine-dependent and machine-independent.
- A compiler can also be thought of as divided into two logical parts (see Fig. 1.11):

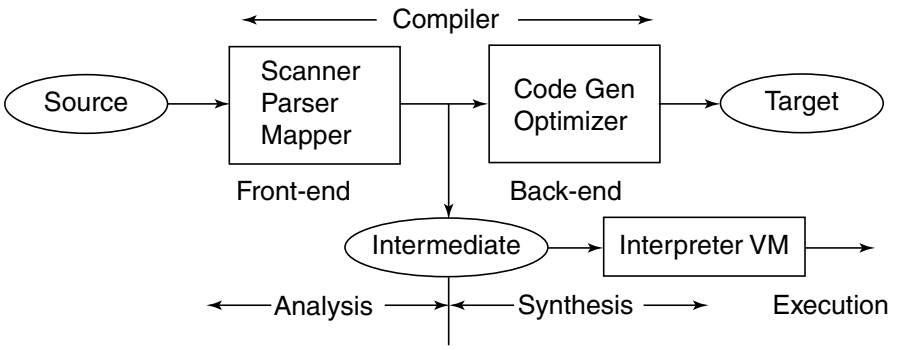


Fig. 1.11 Front- and back-end of a compiler

Front-end of compiler: Consisting of pre-processing, lexical, syntax and semantic phases. This is the part which does the analysis of the input code.

Back-end of compiler: Code generation, Optimization phases. This is the part which does the synthesis of the output or target code.

On the basis of this concept of two parts of a compiler, we can envision several situations for a compiler development project. We may be required to develop compilers for:

- single source language, single target language (1×1);
- multi-source language, single target language ($m \times 1$);
- single source language, multi-target language ($1 \times n$);
- multi-source language, multi-target language ($m \times n$).

In the 1×1 case, we need to develop only one compiler, with one front-end and one back-end. For cases $m \times 1$ and $1 \times n$, we have to develop m and n compilers, respectively. Instead, we really need to develop only m front-ends and one back-end for $m \times 1$ case. Similarly, we really need to develop one front-end and only n back-end for $1 \times n$ case (see Fig. 1.12).

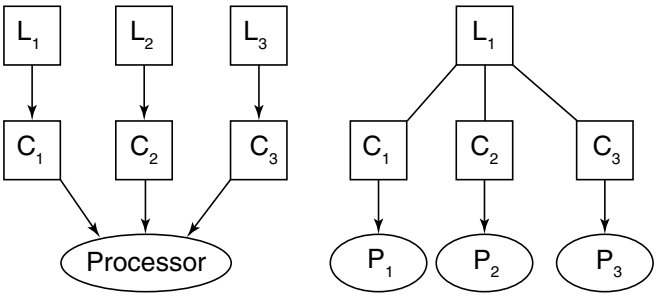


Fig. 1.12 Multiple source, single target ($m \times 1$) and single source, multi-target ($1 \times n$) cases. L Source language, C Compiler, P Processor

The real impact of this idea will be realized in the case of $m \times n$, as shown in Fig. 1.13. Instead of developing $m \times n$ compilers, we need to develop only m front-ends and n back-ends, to fabricate a total of $m \times n$ compilers.

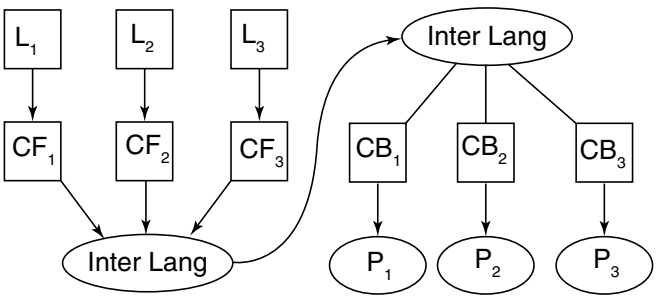


Fig. 1.13 Multi-source, multi-target case ($m \times n$). CF Compiler front-end, CB Compiler back-end

An example of implementation of this idea is the GNU C compiler, some preliminary details of which are given in Section 1.7.

1.6 A More Detailed Look at Phases of a Compiler

We have already seen that a compiler has the following phases:

- Pre-processing
- Scanner – Lexical Analysis
- Parser – Syntax Analysis
- Mapper – Semantic Analysis
- Code Generation
- Assembler
- Optimization (spread over several phases)
- Error Checking (spread over several phases)

A compiler generates intermediate files between these phases, to communicate output of one phase as input to the next (see Fig. 1.14).

We now discuss each phase of a compiler in some details.

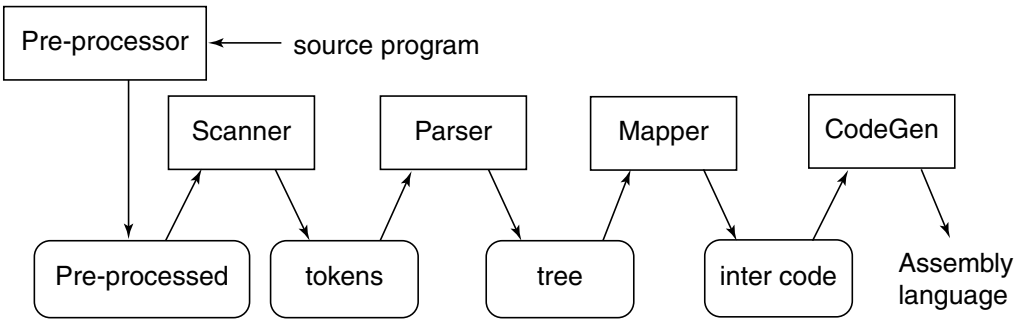


Fig. 1.14 Phases of a compiler and intermediate outputs

1.6.1 Lexical Analyzer – Scanner

Does lexical analysis, by doing the following:

- Analyze individual character sequences and find language tokens, like NUMBER, IDENTIFIER, OPERATOR, etc. Usually, these tokens are internally denoted by small integers, e.g. 257 for NUMBER, 258 for IDENTIFIER, 259 for OPERATOR, etc.
- Send a stream of pairs (token-type, value) for each language construct, to the Parser.
- Lexical tokens are generally defined as regular expressions (RE), for example:

```
NUMBER = [+|-]d*[.]d+
```

- Usually based on a finite-state machine (FSM) model.

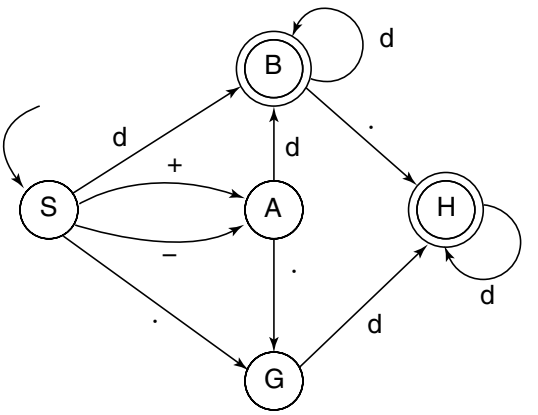


Fig. 1.15 A finite-state machine for example Scanner

Refer to Appendix A for more details. Figure 1.15 shows FSM for the example RE given above.

See the example lexical analyzer code `scanner.c` given below.

```
/*----------------scanner.c----------------*/
#include <ctype.h>
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int token;

next(){
      token = getchar();
}
error(char *s){
   printf("\n error: %s \n", s);
   for(next(); token != '\n'; next());
}
func0(){
   next();
   if(isdigit(token))
      func2();
   else {
      if((token == '+') || (token == '-'))
         func1();
      else {
         if(token == '.')
            func3();
         else
            error("NAN");
      }
   }
}
func1(){
   next();
   if(isdigit(token))
      func2();
   else {
      if(token == '.')
         func3();
      else
         error("Digit or . expected");
   }
}
func2(){
   next();
   if(isdigit(token))
      func2();
   else {
      if(token == '.')
         func4();
      else
         func5();
```

```
    }
}
func3(){
    next();
    if(isdigit(token))
        func4();
    else
        error("Digit expected");
}
func4(){
    next();
    if(isdigit(token))
        func4();
    else
        func5();
}
func5(){
    if(token == '\n') {
        printf("\nNUMBER\n");
    } else
        error("invalid");
}
main(){
    while(1){
        func0();
    }
}
```

This Scanner code is for the finite-state machine shown in Fig. 1.15.

1.6.2 SyntaxAnalyzer – Parser

Does syntax analysis, by doing the following:

- Analyze the stream of (tokens, value) pairs and find language syntactic constructs, like ASSIGNMENT, IF-THEN-ELSE, WHILE, FOR, etc.
- Make a syntax tree for each identified construct.
- Detect syntax errors while doing the above.

Most of the programming languages are designed to be context-free languages (CFL). A push-down automaton, i.e. an FSM + stack is an acceptor for these types of languages. See Appendix A for details.

Consider the following example grammar:

```
S -> E #
E -> T { + T }
T -> F { * F }
F -> lp E rp | n
```

This grammar generates a language consisting of arithmetic expressions like

```
n n+n n*n n+n*n n*n+n (n+n)*n (n+n)*(n+n) ... ...
```

where ‘n’ represents a number token and ‘lp’and ‘rp’, respectively, represent left and right parentheses tokens coming from the Scanner.

The code for a simplified example parser, `parser.c`, for the above grammar is given below. This type of parser is called a ***recursive-descent parser***, whose design is discussed in Chapter 4.

```
/*----------------parser.c----------------*/
#include <stdio.h>
#include <string.h>
int err, t;
char str[82];

error(int i){
   err = 1;
   printf("error number %d\n", i);
   mungetc();
   while(mgetch() != '\n');
   mungetc();
}
nextsymbol(){
   int c;
   c = mgetch();
   mungetc();
   return(c);
}
pE(){
   pT();
   while(nextsymbol() == '+') {
   pplus();
   pT();
   }
}
pT(){
   pF();
   while(nextsymbol() == '*') {
   past();
   pF();
   }
}
pF(){
   switch(nextsymbol()) {
   case '(':{
      plp();
      pE();
      prp();
      break;
   }
   case 'a':{
      pa();
      break;
```

```
    }
    default: error(10);
    }
}
pa(){
    if(symbol() != 'a')
    error(1);
}
plp(){
    if(symbol() != '(')
    error(2);
}
prp(){
    if(symbol() != ')')
    error(3);
}
pplus(){
    if(symbol() != '+')
    error(4);
}
past(){
    if(symbol() != '*')
    error(5);
}
symbol(){
    int c;
    c = mgetch();
    return(c);
}
mgetch(){
    char c;
    t++;
    c = str[t];
    return((c >= 'a' && c <= 'z')?'a':c);
}
mungetc(){
    t--;
}
main(){
    int c;
    while(1) {
        err = 0;
        t = -1;
        fgets(str,82,stdin);
        printf("\n%s", str);
        pE();
        c = mgetch();
        if(c == '\n') {
            if(!err) {
                printf("\n ACCEPTED\n");
```

```
            } else {
                printf("\n REJECTED\n");
                t = -1;
                fgets(str,82,stdin);
                printf("\n%s", str);
                err = 0;
            }
        }
    }
}
```

Note that this simplified parser does not generate the syntax tree, it only detects correctness of syntax.

1.6.3 Semantic Analyzer – Mapper

It converts or maps syntax trees for each construct into a sequence of intermediate language statements.

For example, source:

```
d = a * (b + c);
```

The syntax tree generated by the Parser may look like as shown in Fig. 1.16.

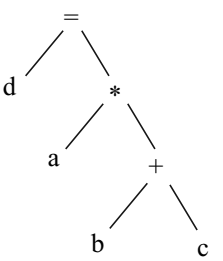


Fig. 1.16 A syntax tree for example statement

The (unoptimized) intermediate language code may look like

```
load   b
add    c
store  t1
load   a
mult   t1
store  t2
load   t2
store  d
```

1.6.4 Code Generation and Machine-dependent Optimization

From the intermediate code, assembly language statements can be generated. Instead of temporaries (t1, t2, etc.) in memory, CPU registers may be assigned.

Code Optimization

In the example intermediate code, several possibilities are there:

- unnecessary stores and loads
- sequence of statements

A better intermediate code could be:

```
load b
add c
mult a
store d
```

1.6.5 Optimization

What we have done above is Optimization. There are two types of optimization – machine-independent and machine-dependent optimizations. There are several possibilities of optimization:

- register allocation,
- taking invariant code outside loops,
- operator rank reduction (e.g. add instead of mult),
- constant expression calculation (e.g. $a = 3.142 * 56$, RHS done at compile time),
- removal of dead code,
- rolling out a loop, etc.

Several types and levels of optimization are available in GNU C compiler, as briefly indicated in Section. 1.7.

1.6.6 How to Develop Optimized Code?

- First get your code working completely error-free without any optimization.
- You should put the code to be optimized (i.e. most time-consuming functions) in a separate source file.
- Be aware of general nature of the optimization (code changes) introduced by different switches.
- Start with lowest level of optimization (-O only), get the assembly output (use -S switch).
- Carefully inspect it; does it do your job still, the way you wanted?
- If yes, apply next higher level of optimization and repeat.

1.7 A Real-life Compiler – `gcc`

The GNU C compiler `gcc` is arguably the best compiler available for C language. In fact it is a suite of compilers, as the package contains `g++` – a compiler for C++, and facilities to compile other languages like Objective-C and Objective-C++, FORTRAN in both fixed form and free form (i.e. F90 and F95 versions), ADA, Java. It can handle different dialects of C.

It is also able to generate executable code for the following CPU families: ARC, ARM, AVR, Darwin, DEC Alpha, HPPA, i386 and x86-64, IA-64, Motorola M680x0, MIPS, PDP-11, PowerPC, RS/6000, S/390, SPARC, TMS320C3x/C4x and others.

Thus, `gcc` is a good example of the ideas presented in Section 1.5.

When you invoke GCC, it normally does pre-processing, compilation, assembly and linking. The "overall options" allow one to stop this process at an intermediate stage. For example,

-c Compile or assemble the source files, but do not link. The linking stage is not simply done. The ultimate output is in the form of an object file for each source file.
-S Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.
-E Stop after the pre-processing stage; do not run the compiler proper. The output is in the form of pre-processed source code, which is sent to the standard output.
-o file Place output in file `file`.
-wall A warning option – displays all possible warnings, very useful during initial debugging of a code.

In fact, `gcc` has a large complement of options for warnings, debugging, optimization, pre-processor, linker, assembler and target machine/language. You should see the man-page of `gcc` for full details. In general, the processing steps followed by `gcc` are:

- pre-processing (invokes `cpp`),
- compilation,
- assembly (invokes `as`), see Fig. 1.17,
- linking (invokes `ld`), see Fig. 1.18.

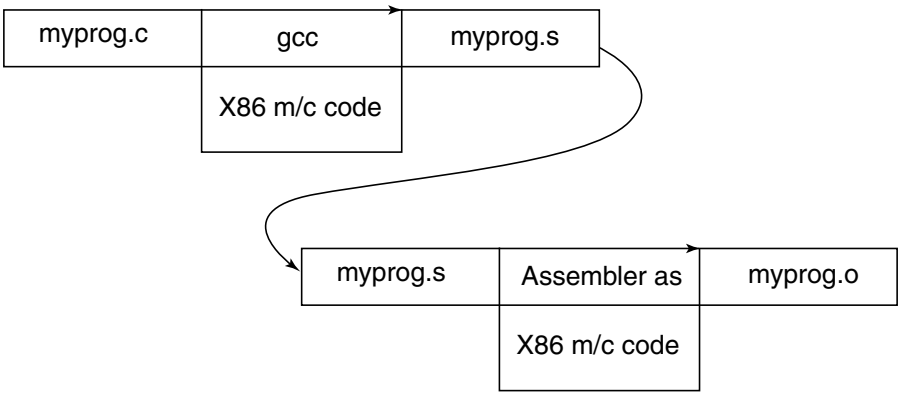


Fig. 1.17 `gcc` converts from C to obj using assembler

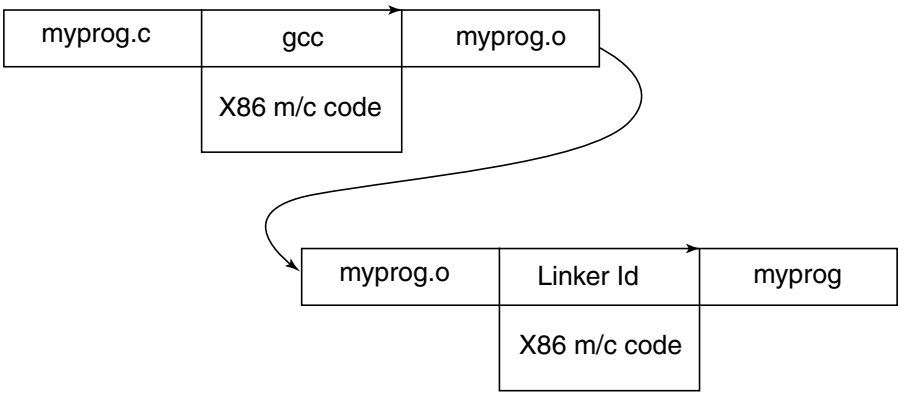


Fig. 1.18 `gcc`: The linker converts an obj file to an executable

In gcc, the level of attempted optimization controlled by `-O` and other switches, but you have to be careful, specially for embedded and Device-driver codes.

In the subsequent chapters, we shall refer to GCC for indicating how relevant phases of a theoretical compiler get implemented in a real-life compiler suite.

1.8 What Do We Mean by "Meaning"?

In this chapter and also the rest of the book, we have used and shall use phrases like: "the machine *understands* a particular language" or "the semantic analysis phase of a compiler finds out the *meaning*

of statements of the input code". It should be clearly understood that this anthropological (human-like) characterization of what a machine is actually doing is for convenience only. The machine does not "understand" the "meaning" of what you, as a programmer, have specified. Perhaps this humanization of the machine behaviour comes from fond attachment to one of our most complex creations – a modern computing machine.

Suppose that you have visited Japan, but do not know Japanese language. Having enough money, you employ a Japanese interpreter, who will help you translating your work in Japan. You go to a shop with her to buy a few shirts. You convey your request to the interpreter and she translates it and conveys it to the shopkeeper. Not only the shopkeeper now knows your request but *the interpreter also* – she "knows" that you want to buy shirts. We say she internalizes the meaning of your request.

If, on the other hand, you carried with you an electronic pocket dictionary available now-a-days, which will actually speak out the Japanese version of input English sentences, that gadget will not "know" what you requested. It simply converts mechanically, without internalizing the meaning of what you said.

Take another example. Suppose your teacher writes on the black-board the following C statements:

```
x1 = (-b + sqrt(b * b - 4 * a * c))/(2 * a);
x2 = (-b - sqrt(b * b - 4 * a * c))/(2 * a);
```

you will most probably understand that he is talking about solutions of a quadratic equation $ax^2 + bx + c$, apart from understanding what actual operations are being specified. To you the statements fit into a larger set of information. As far as a language translator is concerned, it has function to convert each on the arithmetical operations such as +, −, *, /*sqrt* into corresponding actions on the target machine that is all. It has no notion that these statements are solution of some equations, in fact, it does not "understand" words like "equation" or "solution".

Thus when we say that a particular portion of a compiler finds out "meaning" of input code, we only mean that mechanisms to convert that code to corresponding target language construct is available within the compiler, i.e. how to specify corresponding actions in the target language. It is designed to detect a certain set of patterns and map them into another set of patterns. There is no claim that it is able to "understand" the "meaning" of any of those patterns.

Though some amount of success has been obtained in developing computer programs which find out "meaning" of given text or image input, this area of computer technology is still at a very nascent stage and definitely not yet included in the present day language translators.

LOOKING FORWARD

This chapter should have given you an overall idea about a language translator and specifically about a compiler. In the subsequent chapters, we fill in a lot of details.

After going through a step-by-step discussion of a complete compiler and *run-time system* for a very simple language in Chapter 2, we shall discuss in individual chapters each of the phases of a compiler. We shall also discuss some of the concepts that we have not yet mentioned in this chapter – syntax-directed translation, attribute grammars, type system, run-time environment, code generation and machine-dependent optimization, etc.

We shall also introduce gradually compiler writing tools such as `lex, flex, yacc, bison,` etc. and use them wherever necessary. We have implemented many of the programs discussed in the subsequent chapters; their C source code is available as mentioned elsewhere.

HISTORICAL NOTES

Since the beginning of the computer era, the development of computer programming languages and their translators have been driven by advances in hardware technologies on one hand and expectations in terms of range, performance – both correctness and efficiency – and complexity of applications on the other hand. The introduction of the following major concepts in programming languages can be linked to development of 1st, 2nd, 3rd and 4th generations of hardware:

- **1940s:** Assembly language;
- **1950s:** First HLLs: FORTRAN, LISP and COBOL, also ALGOL-60 report – block structuring, identifier scopes;
- **1960s:** BNF used to specify a programming language grammar, APL, BASIC and PL/I developed, though APL and PL/I could not become popular;
- **1970s:** Simula (first object-oriented programming language), C (system programming), SmallTalk (OOP), Prolog (logic programming), ML (type system, functional programming); *Structured Programming*;
- **1980s:** C++ (OO and system programming), ADA (systems programming), in Japan 5th generation language development, modular programming with parameterized modules, emergence of RISComputers emphasized compiler orientation in CPU design; Perl and Eiffel developed;
- **1990s and later:** Programmer productivity becomes the driving force – VisualBasic, C# developed; Java (OOP and built-in garbage collection), Python, Ruby, JavaScript, XML;
- **Current trends:** Component-oriented software development, mechanisms for security and reliability verification in the language, integration with database, further development of Web languages.

See a Wikipedia article at:
`http://en.wikipedia.org/wiki/History_of_programming_languages`

O'reilly has published a very interesting language history poster, see:
`http://oreilly.com/news/languageposter_0504.html` and
`http://oreilly.com/news/graphics/prog_lang_poster.pdf`

EXERCISES

1. Make a list of all the programming languages that you can find and make a table for them indicating (i) if they are interpreted, compiled or part-compiled, (ii) the platforms on which available and (iii) application areas in which used. Do not forget languages for pictures generation (e.g. `povray`), printing (e.g. `postscript`), web (e.g. HTML, Javascript), etc. After you have made this table, compare it with Fig. 1.1.
2. Certain properties of a programming language may require that the only way to get the code written in it be executed is by interpretation. In other words, compilation to a native machine code of a traditional CPU is not possible. What are these properties? Try to investigate in as much detail as you can. [Hint: Consider the way an interpreter executes a code and the way a compiled program is executed.]
3. Look at source `optimise1.c`. Study `optimize10.s`, `optimize100.s`, `optimize101.s`, `optimize102.s`, `optimize103.s`, `optimize103s.s`. Which of these optimizations will be unacceptable in some applications or situations?
4. Obtain details of METAFONT and METAPOST languages from Internet and study the basic objects, operations and functions provided in them.

5. Portability of a program is said to be a desirable feature made available by an HLL. What issues can arise in making a program truly portable, i.e. one should be able to compile it and successfully execute it on any family of CPU?
6. Suresh wrote the following small program in C, to find out execution speed of a particular CPU:

```
#include <stdio.h>
int main(){
  int i, a;
  for(i = 0; i<100000000; i++){
    a = 1;
  }
}
```

He compiled it normally using `gcc testopt.c -o testopt`, executed it and got a running-time estimate of 0.288 s on an average. Then he recompiled it with `gcc -03 testopt.c -o testopt` (i.e. level 3 optimization) and got a running-time estimate of 0.003 s on an average. Surprised at this large reduction in time, he obtained the assembly language output for both the versions of translation:

```
# unoptimized
main:
       leal   4(%esp), %ecx
       andl   $-16, %esp
       pushl  -4(%ecx)
       pushl  %ebp
       movl   %esp, %ebp
       pushl  %ecx
       subl   $16, %esp
#----------------------------------
       movl   $0, -12(%ebp)
       jmp    .L2
.L3:
       movl   $1, -8(%ebp)
       incl   -12(%ebp)
.L2:
       cmpl   $99999999, -12(%ebp)
       jle    .L3
#----------------------------------
       addl   $16, %esp
       popl   %ecx
       popl   %ebp
       leal   -4(%ecx), %esp
       ret

# optimized
main:
       leal   4(%esp), %ecx
       andl   $-16, %esp
       pushl  -4(%ecx)
       pushl  %ebp
```

```
        movl   %esp, %ebp
        pushl  %ecx
#----------------------------------
        popl   %ecx
        popl   %ebp
        leal   -4(%ecx), %esp
        ret
```

Comparing the two codes he found that the loop forming instructions, marked by two horizontal lines in the unoptimized code, were completely missing in the optimized code. Also, the local variables `i` and `a` were not created. What was going on?

WEB RESOURCES

The Catalog of Compiler Construction Tools `http://catalog.compilertools.net/` – Freeware and Commercial resources for compiler writers.

GLOSSARY

Language A systematic means of communicating ideas or feelings by the use of conventionalized signs, sounds, gestures or marks having understood meanings; the means by which animals communicate; a formal system of signs and symbols (as FORTRAN or a calculus in logic) including rules for the formation and transformation of admissible expressions.
Internalize (UK internalize) To accept or absorb an idea, attitude, belief, etc. so that it becomes part of your character.
Code An algorithm expressed in some programming language.
Compiler A language translator which converts an HLL program to the native machine language of a particular computing machine.
Source The input program to a translator.
Machine It is a device to do some job. It can be *abstract* or *actual*, and if actual, implemented as a program (a virtual machine) or as a physical hardware. A machine accepts and executes *commands* or *instructions*.
Program A list of commands to some machine, to do a specific job.
High-level Language A language for writing computer programs which looks more like human language than computer's native machine language and is therefore easier to understand.
Target The language of the machine or system on which the user program will be ultimately executed.
Token The Scanner detects various language atoms in the source code and represents them in the form of a uniform sized tokens, for ease of processing by the subsequent phases.
Phase Part of processing cycle during which a well-defined transformation of the input is made.
Optimization Systematic attempt by compiler to generate a final output, in the target language, which is a good approximation to what an experienced programmer of the target language would have written.
Statement A unit of specification of a complete action – which may be *declarative* or *imperative*.
Macro An identifier which specifies a portion of code. During *macro expansion*, a macro will be replaced by that code. Tip of an iceberg.
Meaning The actions, expressed in the target language, specified by a portion of the source code.
Assembly Conversion of an assembly language program into machine language program.

Parser – Syntax analyzer Identifies various language constructs, which determine the statement types and build the *parse tree* representation of the input program.
Scanner – Lexical analyzer Identifies various language atoms, maps them to tokens and passes them on to parser.
Formal A representation based on strict rules for simple, definitive, unambiguous and mathematical description of any system.
Execution Doing the computational work specified by the program.
Load-and-Go A scheme of HLL use, wherein the compiler, source code and the target code all reside in the machine memory simultaneously.
Interpreter An execution agent – which may be a software or hardware – which seemingly execute an HLL program directly. It usually has to interpret each statement just before its execution.
Syntax The grammatical structure of a program.
Instructions Commands to a computing machine.
Debugging Removal of "bugs" – errors – from a program.
Lexical Concerning the way various language atoms are expressed in terms of a specific character set.
Semantic Concerning the *meaning* of various statements in a program.
Loader A utility program which loads an executable program file on a secondary storage into the main memory of the machine. Depending upon the platform, it may have to do certain modifications to the code.
Linker A utility program which sets up address links between various parts and modules of a translated program and also any program libraries to be used.

2

A Simple Translator

What you will learn in this chapter

- Grammar of a simple high level language
- A lexical analyzer for that language
- A Recursive-descent Parser
- Intermediate code
- How to generate code for a target machine?
- A target virtual machine
- Need for optimization
- Simulator for the virtual machine

Key Words

`simple`, Scanner, parser, RDP, intermediate code, virtual machine, simulator, optimization

In this chapter, we discuss a very simple C-like language called `simple` and its compiler. Though due to pedagogic necessity the language is kept as simple as possible, it will illustrate all the significant issues of a compiler construction. Concentration will be on exposing the compiler writing steps rather than power of the language being compiled. The language can be extended to some extent, to make it more "practical".

We shall carry our discussion up to and including code generation. This requires specification of the target language. We have selected to have a simple target language for a virtual machine, VM1. A real-life processor has a highly optimized instruction set, which is not suitable for introducing and illustrating basic ideas. Our virtual machine simulator displays information useful for understanding what is going on inside it while a program is being executed.

2.1 A Simple Language

We define a simple – but high enough level – language as an example. At present it contains just sufficient construct to write small programs, which can be compiled with the compiler we are going to discuss and executed on the virtual machine VM1. VM1 is a working computer; it is implemented only in software as a C program and not in silicon. As indicated in Chapter 1, a similar idea is actually used in programming languages such as Java, Perl and Python.

A typical program in `simple` looks like:

```
      program
      int a ;
      let a = 10 ;
loop: print(a) ;
      let a = a - 1 ;
      if a goto loop ;
      end
```

This program will print numbers 10 to 0, in descending order. Some of the language constructs are redundant at this stage (e.g. declaration of a variable as an integer), but we may extend this language "some day" and add floating point quantities, variable list in the `print` statement, etc.

Note the way we have formed the loop in the above example. There are only two *control* statements – `if` and `goto` – in this language. There are even no relational operators.

To have a precise specification of the language, we give its grammar below.

2.1.1 Grammar of `simple`

The grammar could have been given in Backus-Naur Form (BNF), see Appendix A, but in anticipation of our discussions about compiler writing tools like `yacc`, we give the grammar in a format compatible with `yacc` input language.

In the `yacc` format, the Non-Terminals or *meta-variables* are written as identifiers and the Terminals or *meta-literals* are written within single quotes. Alternate productions are separated by vertical bar '|', the non-terminal on LHS is followed by a colon ':' and the productions for a particular non-terminal are terminated by semi-colon ';'.

```
program : 'program' statements 'end'
        ;
statements : statement
           | statements statement
           ;

statement : basic-statement ';'
          | label basic-statement ';'
          ;
basic-statement : assignment
                | var
                | if
                | goto
                | read
                | print
                ;
assignment : 'let' variable '=' expr
               ;
var : 'int' variable
      ;
if : 'if' expr statement
     ;
```

```
goto : 'goto' variable
       ;
read : 'read' '(' variable ')'
       ;
print : 'print' '(' number ')'
      | 'print' '(' variable ')'
      ;

expr : expr '+' term
       | expr '-' term
       | term
       ;

term : term '*' factor
     | term '/' factor
     | factor
     ;
factor : '(' expr ')'
       | variable
       | number
       ;
label : variable ':'
      ;

variable : char
           | variable char
           | variable digit
           ;
number : '+' unsigned-number
         | '-' unsigned-number
         | unsigned-number
         ;
unsigned-number : digit
          | unsigned-number digit
          ;
          ;
char : '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'
      ;
op    : '+'|'-'|'*'|'/'|'('|')'|':'|'='|';'|'<'|'>';
```

A close look at the grammar will tell you that each statement type starts with a *keyword*. Such a design simplifies the job of writing the parser for this language.

Note that the grammar does not specify certain semantic requirements, for example, a valid program in `simple` should have a label, say, `loop` given somewhere, if there is a `gotoloop` in the program. Similarly, we require that all the variables be declared before use, the `print` or `read` statement cannot have a label as their argument, etc. The grammar does not specify these things, but a compiler will have to check for such unacceptable constructs.

2.1.2 Target Language

We shall now define the target language, i.e. the language understood by our target machine VM1. Our VM1 is a 32-bit, fixed instruction size, binary computer. The programmer view of the machine is shown in Fig. 2.1

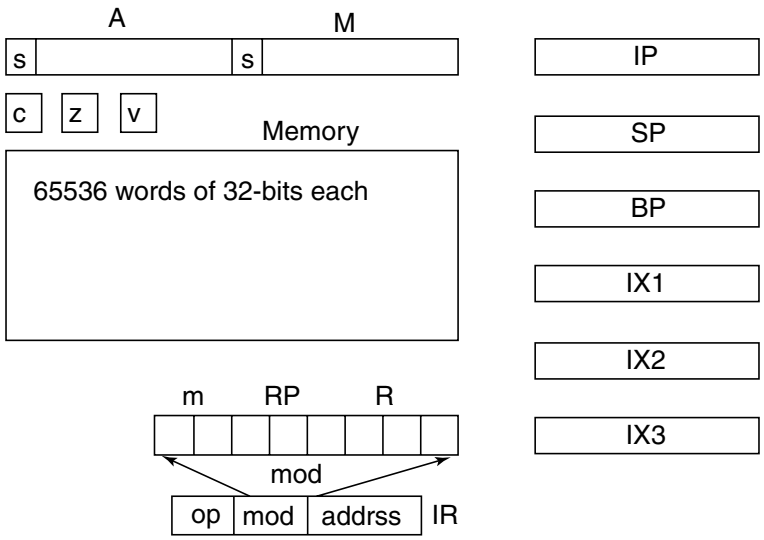


Fig. 2.1 A programmer's view of VM1

The following registers are available:

A – 32-bit accumulator, leftmost bit is interpreted as its sign bit. Register no. 0.
M – Auxiliary register, leftmost bit is interpreted as its sign bit. Register no. 4.
IP – Instruction pointer, 32-bits, but only lower 16-bits significant at present. Register no. 7.
SP – Stack Pointer, 32-bits, but only lower 16-bits significant. Register no. 6.
BP – Base Pointer, 32-bits, but only lower 16-bits significant. Register no. 5.
IX1 – Index Register 1, 32-bits; Register no. 1.

Table 2.1 Some of the op-codes of VM1

Hex	Mnemonic	Hex	Mnemonic	Hex	Mnemonic
00	NOP	10	PUSH	08	JMP
01	ADD	11	ADDI	09	JZ
02	SUB	12	SUBI	0A	JP
03	MUL	13	MULI	0B	JN
04	DIV	14	DIVI	0C	
05	LD	15	LDI	0D	
06	ST	16		0F	
07	LA	17	POP	FF	HLT

IX2 – Index Register 2, 32-bits; Register no. 2.
IX3 – Index Register 3, 32-bits; Register no. 3.
IR – Instruction Register, 32-bits; divided into OP (1-byte), MOD (1-byte) and ADDRS (2-bytes); holds an instruction being executed.
MOD – A part of IR, consisting of m (2-bits), RP (3-bits), R (3-bits); m determines the addressing mode used:

00 – Direct address
10 – Indirect w.r.t RP (the numbered registers R)
01 – Indexed w.r.t RP
11 – Indirect (indexed w.r.t RP)

Some of the op-codes of VM1 are given in Table 2.1.

Address 0000 is for an input/output device. LD 0000 will read from the input device and ST 0000 will print to the output device.

2.1.3 Example Program in Machine Code

We write the example program given above in the "machine code" of VM1:

```
0100  |  15 0 0 0 000A  |  Load value 10 in A-reg
   1  |  06 0 0 0 0110  |  store in a
   2  |  06 0 0 0 0000  |  display (store in 0)
   3  |  05 0 0 0 0110  |  load a
   4  |  12 0 0 0    1  |  subtract 1
   5  |  0A 0 0 0 FFFB  |  Jump if +ve (relative)
   6  |  FF 0 0 0 0     |  Halt
0110  |  XX X X X X     |  value of a stored here
```

Note that the jump instructions take relative jump, from the current value in IP.

Now we shall discuss in some details the compiler for the `simple` for the target language of VM1.

2.2 Compiler for Simple

The compiler will consist of three phases: Scanner, parser and mapper-cum-code generator. The source codes and sample test data are available elsewhere.

2.2.1 Scanner

The Scanner is "hand-coded" instead of using a Scanner generator like `lex` or `flex`. We first separate out the `simple` language grammar in two parts – first which will be used as a specification for writing the Scanner and second, as specification for the parser. This is a kind of "division of labour" between two parts of the compiler, but it also allows the lexical aspects of a language to be separated and concentrated, so that, for example, the same language can be implemented in some other international script like Devanagari.

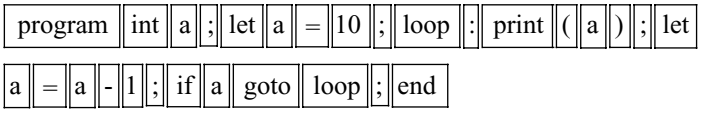
The Scanner-specific part, what we shall call the "Scanner grammar", is given below:

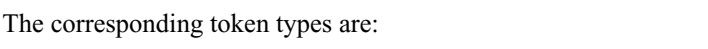
```
variable : char
           | variable char
           | variable digit
           ;
number : '+' unsigned-number
         | '-' unsigned-number
         | unsigned-number
         ;
unsigned-number : digit
         | unsigned-number digit
         ;
char : 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'
     |'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'
     ;
digit : '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
      ;
op    : '+'|'-'|'*'|'/'|'('|')'|':'|'='|';'|'<'|'>';

VAR    : variable ;
NUMBER : number ;
OP     : op ;
```

Note that the last three lines of the Scanner grammar indicate the *tokens* that will be supplied to the parser for further processing. Also, before a sequence of non-whitespace characters is detected as a *variable* and sent to the parser as such, it is compared with a list of *keywords* like `let, if,` etc. and sent to the parser as those keywords.

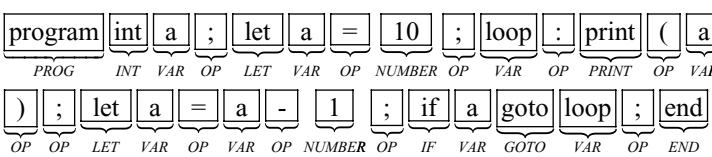
Note that the decision to detect the keywords at lexical level results in a strict rule for not allowing the keywords in any other statement in the source code, for example, you cannot use `if` as a variable name.

The Scanner will have to identify various language *atoms* and generate corresponding *tokens*, which is a one-to-one mapping. For example, the following atom identification should take place for our example `simple` program:





The corresponding token types are:



A token is a pair (*type*, *value*). For the example `simple` program given above, the token sequence generated by the Scanner is

```
(265 0)   (268 0)   (258 0)   (259 59)  (267 0)   (258 0)
(259 61)  (257 10)  (259 59)  (258 1)   (259 58)  (262 0)   (259 40)
(258 0)   (259 41)  (259 59)  (267 0)   (258 0)   (259 61)
(258 0)   (259 45)  (257 1)   (259 59)  (260 0)   (258 0)
(261 0)   (258 1)   (259 59)  (266 0)
```

In our implementation, the token types have positive integer values greater than 256, to maintain compatibility with later discussion of `yacc`. The token type values are declared in the header file `simple.h` as:

```
#define  BASE     256
#define  NUMBER   BASE+1
#define  VAR      BASE+2
#define  OP       BASE+3
#define  IF       BASE+4
#define  GOTO     BASE+5
#define  PRINT    BASE+6
#define  READ     BASE+7
#define  LABEL    BASE+8
#define  PROG     BASE+9
#define  END      BASE+10
#define  LET      BASE+11
#define  INT      BASE+12
#define  MAT      BASE+13
#define  UNDEF    0
```

The tokens are stored as pairs defined by a structure:

```
struct lexv {
  int type;
  union {
    Symbol *sym;
    int value;
  };
};
```

Note the `union` as the token value – for some token types, for example VAR, we will use its assigned address in the memory as the value. This address is represented by a pointer to a corresponding node in a symbol table, formed as a symbol list. The symbol nodes have a structure:

```
typedef struct Symbol
{
  char *name;
  short type;
  short addrs;
  struct Symbol *next;
} Symbol;
```

Some of the fields in such symbol nodes will be filled in by later phases of the compiler – the Scanner fills in only `name`, `type` and `next`.

Whenever a variable is detected by the Scanner, a check is made by the function `lookup()` to see if the identifier is present in the symbol list and if not present, the function `install()` will insert it in the list. The algorithm for `lookup()` is:

Algorithm 2.2.1: lookup(s)

```
sp ← symlist;
while sp not NULL do
    if sp −> name equals s then
        return sp;
    end
    sp ← sp −> next;
end
```

The algorithm for `install()` is:

Algorithm 2.2.2: install (s, t, ad)

```
sp ← memory of size Symbol;
sp −> name ← memory of size of s;
sp −> name ← s;
sp −> type ← t;
sp −> addrs ← ad;
insert sp in symlist as head node;
return sp;
```

The Scanner algorithm is:

Algorithm 2.2.3: Scanner

```
read next character c and ignore white spaces;
if end of file then
    return success;
end
if c alphabetic then
    while alphanumeric and not EOF do
        read and collect characters as string sbuf;
    end
    if sbuf ∈ {int, let, if, goto, print, read, program, end} then
        lexval.value ← 0;
        lexval.type ← corresponding type-value;
        return lexval.type;
    end
    if lookup(sbuf) = NULL then
        install(sbuf,VAR,addrss);
        addrss ++ ;
    end
```

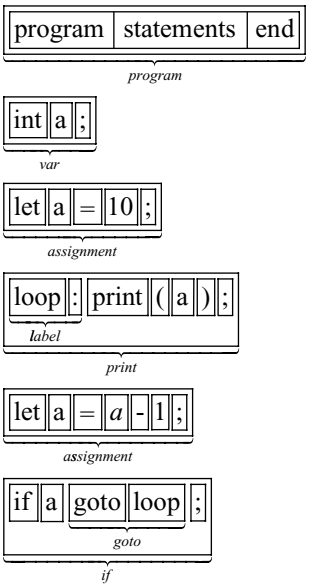
Algorithm 2.2.3: Scanner (*Cont'd*)

```
end
if c numeric then
    read the number into lexval.value;
    lexval.type ← NUMBER;
    return lexval.type;
end
if c ∈ {+− */() : = ; < >} then
    lexval.value ← c;
    lexval.type ← OP;
    return lexval.type;
end
```

2.2.2 Parser

The job of a parser is to identify the syntactic entities in the source code, build and output the parse tree.

For the example `simple` program, the syntactic units should be identified as:



We have opted for implementing the parser for `simple` as a *recursive-descent parser* (RDP). This relatively simple parser implementation is possible if certain restrictions are put on the language and its grammar, see Chapter 4 for details.

The grammar with which the parser is concerned is given below:

```
program : PROG statements END
        ;
statements : statement
           | statements statement
           ;
```

```
statement : basic-statement ';'
          | label basic-statement ';'
          ;
basic-statement : assignment
                | var
                | if
                | goto
                | read
                | print
                ;
assignment : LET variable '=' expr
             ;

var : INT variable
      ;
if : IF expr statement
     ;
goto : GOTO variable
       ;
read : READ '(' variable ')'
       ;
print : PRINT '(' number ')'
      | PRINT '(' variable ')'
      ;

expr : expr '+' term
       | expr '-' term
       | term
       ;

term : term '*' factor
     | term '/' factor
     | factor
     ;
factor : '(' expr ')'
         | VAR
         | NUMBER
         ;

label : VAR ':'
      ;
```

Note that the identifiers written in all capitals are the token types passed on by the Scanner. The operation symbols like '+' are token values for token type OP. The rest are the meta-variables of the grammar.

The basic idea in an RDP is that:

1. For each NT and T symbols, there is a corresponding function which will try to check if the source code at a particular stage of processing can be generated using:
2. if an NT – RHS of productions for that NT;
3. if a T – it confirms that Terminal symbol.

For example, consider the productions for `expr`:

```
expr : expr '+' term
     | expr '-' term
     | term
     ;
```

The above portion of the grammar is in left-recursive form and needs to be converted to right-recursive or right-iterative form:

```
expr : term { '+' term }
     | term { '-' term }
     | term
     ;
```

where the curly brackets '{' and '}' enclose portions which may be repeated one or more times.

The corresponding algorithm `pE()` looks like this:

Algorithm 2.2.4: pE()

```
pT();
while nextsymbol is + or – do
    if nextsymbol is + then
        pplus();
    end
    if nextsymbol is – then
        pminus();
    end
    pT();
end
```

Here, `pT()` is the algorithm, similar to the above, for recognizing all the productions from a `term`. The `pplus()` algorithm is:

Algorithm 2.2.5: pplus()

```
if symbol is not + then
    ERROR;
end
```

The algorithms for `pminus()`, and in fact all the Terminal symbols, are similar.

Notice the simplicity of algorithm – there is a one-to-one correspondence between the grammar production and the algorithm steps. We can write the algorithm almost by inspection.

The corresponding C language implementation should be looked up in the source file `simple2.c` available elsewhere.

Almost the complete parser is written this way, but two functions need special mention here. The parser gets its next token by calling the `Scanner()`. There is a bit of complication here – if you look at the algorithm, for example, for the `pE()`, you will notice that we are checking the *nextsymbol* in order to decide if we should go ahead and collect further `terms` and also if '+' or '−' is to be read

and confirmed. We shall see in Chapter 4 why we need to do this *look-ahead*. We cannot get the `nextsymbol` by requesting `Scanner()`, as the meaning of a look-ahead is that we peep at the next symbol, but without processing it. Thus, we have a two-position "shifter" coded, which is initially empty (see Fig. 2.2).

For each call to `symbol()`, a request goes to the `Scanner` to supply the next token and the shifter is shifted to the left. The very first call to `symbol()` results in two calls to the `Scanner`. A call to `nextsymbol()` does not result in a call to the `Scanner`, it simply allows a peep to the `nxtsym` in the shifter.

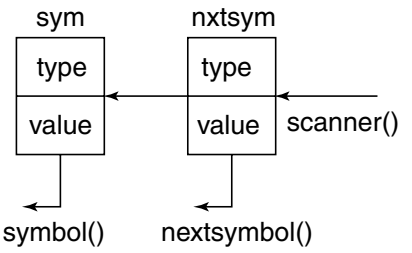


Fig. 2.2 A token shifter used to supply next symbol

The algorithms described till now will just detect the syntax of various statements in the source code. They will decide if the statements are valid or not. Where is the *parse tree* expected to be generated as the output of the parser? In an RDP, it is "invisible", rather it is in the call tree of various function that gets created as each statement is being parsed. For example, consider the following small program.

```
program
let a = 9;
end
```

The corresponding call sequence in the parser is

```
pPROG
  |
pSTS
  |
pST
  |
pLET -> pvar -> pasg -> pE
                        |
                        pT
                        |
                        pF
                        |
                        pnumber
  psemi <---------------+
pEND
```

What we are interested in is an *Intermediate form* of the code, from which we can easily generate the target code, the parse tree itself is not of direct interest. On the other hand, the intermediate form should be such that it facilitates the *optimization* of the target code and tends to give a clean separation of machine-independent and machine-dependent portions of the compiler, for the reasons suggested in Fig. 1.13.

We define the intermediate form we plan to use in the next section.

2.2.3 Intermediate Form and Semantic Phase

There are several possible intermediate forms, as will be discussed in Chapter 8, some of which are tree data structure (Abstract Syntax Tree), a directed acyclic graph (DAG) and N-tuple, of which two-address code (TAC) is an instance. We use here the two-address code method.

The TAC or 3-tuple is a linear sequence or array of 3-component entries, each having a structure shown in Fig. 2.3.

This array is called the *matrix*. The three components have the following significance and use:

op – the "operation" to be done
oprnd 1 – the first "operand"
oprnd 2 – the second "operand"

Each of these components is a token. An additional token type is now defined – MAT, which denotes result obtained when a specific entry is "executed". The token value for MAT denotes the entry number.

Thus, it is seen that this form of code looks some what like machine code of some abstract machine. For our example `simple` program, the intermediate form output is given below:

	op	oprnd 1	oprnd 2
1			
2			
3			
4			
n			

Fig. 2.3 Linear 2-address code array, called *matrix*

```
0: (INT,0)     (VAR,0)  (,0)
1: (LET,0)     (VAR,0)  (NUMBER,10)
2: (LABEL,0)   (VAR,1)  (OP,58)
3: (PRINT,0)   (VAR,0)  (,0)
4: (OP,45)     (VAR,0)  (NUMBER,1)
5: (LET,0)     (VAR,0)  (MAT,4)
6: (IF,0)      (VAR,0)  (,0)
7: (GOTO,0)    (VAR,1)  (,0)
8: (FI,0)      (,0)     (,0)
9: (END,0)     (,0)     (,0)
```

The meaning of this sequence is:

entry 0 op = INT which denotes that an integer is being declared; oprnd1 = address assigned to variable 'a'. oprnd2 is unused.
entry 1 op = LET which denotes that an assignment is to be made; oprnd1 is the address of 'a'; oprnd2 is the RHS value to be assigned.
entry 2 op = LABEL: set a label on this entry, oprnd1 = label 'loop'.
entry 3 op = PRINT: a value to be printed; oprnd1 = address of 'a'.
entry 4 op = OP, value is '−', i.e. minus operation; oprnd1 is 'a'; oprnd2 is number 1. This will execute as $a - 1$ and the result will be known as MAT,4.
entry 5 op = LET: assignment to oprnd1 = 'a', value assigned is oprnd2 = MAT,4, i.e. result of the entry 4. Effectively, $a = a - 1$.
entry 6 op = IF: oprnd1 is 'a'; if value of oprnd1 is non-zero, do the next entry and subsequent entries till op = FI.
entry 7 op = GOTO: oprnd1 is label 'loop'.
entry 8 op = FI: end of "range" of the IF predicate.
entry 9 op = END: end of the program.

How does this intermediate code get generated? We can argue like this: the very fact that the RDP takes a recursive descent for a statement indicates that something is to be done first, before doing whatever is at hand. Thus, the code is generated and inserted in the matrix while the parser is *ascending* out of the descent. For details about the order of execution of statements within a recursive function, you may refer to the section on "Execution Trace and recursive functions", under Section A.1.8.

Thus, the trick is to put all the code generation and insertion steps *after* a recursive call, to make it executed during the ascent.

For example, consider a statement

```
let a = b + c * d ;
```

The algorithm segment for a LET statement is:

Algorithm 2.2.6: pLET() function

```
t ← (LET,0);
symbol();
t1 ← pvar();
pasg();
t2 ← pE();
insert in matrix (t, t1, t2);
```

The parser will detect an assignment statement (due to 'let'), look for a variable, encounter 'a' and its token is assigned to `t1`. Then it looks for and confirms the '=' with `pasg()`. Then it invokes `pE()`, the function to recognize all the substrings which can get produced out of non-terminal *expr*. That function will parse the RHS expression and generate and insert in matrix whatever the 2-address code is needed to calculate the expression. The final result of the calculation will be represented by a particular matrix line, say *n*, and token (MAT, n) will be returned by `pE()`, which is assigned to `t2`. As the last step, an entry (t, t1, t2) will be inserted in the matrix, which specifies assignment (LET) to specified variable, the result of expression detected by `pE()`. Note very well that all the matrix entries needed to calculate the RHS expression are generated and inserted in the matrix, *before* the assignment operation entry is inserted.

The extended algorithm for `pE()` is:

Algorithm 2.2.7: pE() – returns a token

```
m ← −1;
t1 ← pT();
while nextsymbol is + or − do
    if nextsymbol is + then
        t2 ← pplus();
    end
    if nextsymbol is – then
        t2 ← pminus();
    end
    t3 ← pT();
```

Algorithm 2.2.7: pE() – returns a token (*Cont'd*)

```
    if m = −1 then
        t ← t1;
    else
        t ← (MAT,m);
    end
    m ← matrix(t2, t, t3);
end
if m = −1 then
    t ← t1;
else
    t ← (MAT, m);
end
return t;
```

We have already encountered this algorithm above, but here we have added a few more steps to generate the 2-address code. The algorithm for `pT()` is similar. The algorithm for `pF()` is given as Algorithm 2.2.8.

Algorithm 2.2.8: pF( ) – returns a token

```
switch nextsymbol( ) do
    case OP
        if nxtsymval = '(' then
            plp( );
            t = pE( );
            prp( );
        else
            ERROR;
        end
    end
    case NUMBER
        t = pnumber( );
    end
    case VAR
        t = pvar( );
    end
end
return t;
```

Note that `pF()` itself does not insert any entries in the matrix. The 2-address code generated is:

```
0:(OP,42)     (VAR,2)  (VAR,3)
1:(OP,43)     (VAR,1)  (MAT,0)
2:(LET,0)     (VAR,0)  (MAT,1)
```

The matrix line 0 specifies a '*' operation on 'c' and 'd', line 1 specifies a '+' operation on 'b' and result of line 0, and line 2 specifies '=' operation on 'a' with result of line 1 as the RHS. You should trace through the algorithms for `pE()`, `pT()` and `pF()` and satisfy yourself that the above matrix will be obtained.

Now we discuss how the target language code, for our machine VM1, is generated.

2.2.4 Code Generation

Once we have the linear 2-address intermediate code, code generation for VM1 is comparatively straightforward. We have a separate program `codegen.c` for this operation. It accepts the output from parser-cum-semantic routines in `simple2.c` and generates the "machine language" code for VM1. Our test program source code is:

```
program
int a ;
let a = 10 ;
loop: print(a) ;
let a = a — 1 ;
if a goto loop ;
end
```

The Scanner-parser stages gave us an intermediate form output (in numeric form):

```
0  268  0   258 0  0    0
1  267  0   258 0  257  10
2  264  0   258 1  259  58
3  262  0   258 0  0    0
4  259  45  258 0  257  1
5  267  0   258 0  269  4
6  260  0   258 0  0    0
7  261  0   258 1  0    0
8  270  0   0   0  0    0
9  266  0   0   0  0    0
```

which is equivalent to intermediate code in alphabetic form:

```
0:(INT,0)    (VAR,0)  (,0)
1:(LET,0)    (VAR,0)  (NUMBER,10)
2:(LABEL,0)  (VAR,1)  (OP,58)
3:(PRINT,0)  (VAR,0)  (,0)
4:(OP,45)    (VAR,0)  (NUMBER,1)
5:(LET,0)    (VAR,0)  (MAT,4)
6:(IF,0)     (VAR,0)  (,0)
7:(GOTO,0)   (VAR,1)  (,0)
8:(FI,0)     (,0)     (,0)
9:(END,0)    (,0)     (,0)
```

When this intermediate code is given to `codegen` it produces the VM1 "machine code" output:

```
140 0 0 0 0 0      | place for 'a'
100 15 0 0 0 a     | LDI   A,   10
101 6 0 0 0 140    | ST    A,   a
102 5 0 0 0 140    | LD    A,   a
103 6 0 0 0 0      | ST    A,   0       (print)
104 5 0 0 0 140    | LD    A,   a
105 12 0 0 0 1     | SUBI  A,   1
106 6 0 0 0 204    | ST    A,   Temp4
107 5 0 0 0 204    | LD    A,   Temp4
108 6 0 0 0 140    | ST    A,   a
109 5 0 0 0 140    | LD    A,   a
10a b 0 0 0 0      | JN    A    ????   (Jump if -ve to ?)
10b 8 0 0 0 fff6   | JMP   -10         (jump to 102)
10a b 0 0 0    1   | JN    A    +1     (jump if -ve to 10C)
10c ff 0 0 0 100   | HLT               (start address = 100)
```

For your reference, an "assembly language" comments are given, though not part of the code.

We have a number of comments on this code, but we postpone them till we first discuss how this code is generated.

The code generator uses a rather simple method to generate the VM1 code. Note that we have not used the full capabilities of VM1 while generating the code – only one machine register A is used as an accumulator, and various addressing modes like indexed, indirect, etc. are not used. As the `simple` language does not provide subroutines, we do not need to use the SP and BP registers either.

For each type of operation in the 2-address intermediate code, the code generator outputs one or two VM1 instructions. They are listed in Table 2.2.

Three VM1 memory address counters and some data areas are maintained by the `codegen()`:

txtadrs: Address where the next generated VM1 instruction will be positioned. This counter is incremented on each generated instruction.
dataadrs: Address where some data specified by `int` declarations are positioned. This counter is incremented on each generated data area allocation.
matadrs: Address where temporary locations, known as the MAT locations, used during the intermediate code generation, are located.
stack: where the jump address used to implement IF and GOTO are pushed and popped.
vartab: An address array where the assigned VM1 memory address of each of the variables is stored.

The LABEL, GOTO, IF and FI require a somewhat special treatment, because they can result in a forward jump in the target code. The forward jumps are managed through use of an address stack. The semantics of our `if` statement requires that if the value of the expression is non-negative, execute the sub-statement, otherwise go to the next statement. This means we have to insert a Jump-if-Negative instruction when an `if` is encountered. Unfortunately, we do not know the address to which this jump should take place, as the matching FI may be several instructions in the forward direction. We insert a zero address JN instruction and remember where it is positioned by pushing its address on a stack. Later, when an FI is encountered, we pop the stack, and replace the old JN instruction with a new one, with a proper address. As our loader for VM1 is a random address loader, this can be easily achieved.

The possibility of a forward LABEL is dealt with in a simple way. At the beginning of the `codegen()`, a scan of the intermediate code is performed, and when a LABEL is encountered we save the VM1 memory address where it occurred in the `vartab`. This is possible to do, as we know how many instructions are generated for each type of operation in the intermediate code. These lengths are stored in a data array `instrlen`. Later, when a GOTO is encountered with that identifier as the label, we simply generate a JMP instruction with that saved address. The jump addresses are adjusted with respect to the current instruction address, as required by the semantics of VM1.

The action of `codegen` for remaining intermediate code operations is rather straightforward one-to-one mapping and should be clear from Table 2.2.

Table 2.2 Mapping from Intermediate code to VM1 code

Intermediate	VM1 machine code	`codegen` actions and remarks
INT opr1	dataadrs,0,0,0,0,0	loader will reserve and clear this location. vartab[opr1] ← dataadrs dataadrs++
LET opr1 opr2	txtadrs, LD/LDI,0,0,0,opr2	LD if opr2 is VAR or MAT LDI if opr2 is NUMBER opr2 = matstart + opr2 or datastart + opr2 txtadrs++
	txtadrs, ST,0,0,0,opr1	opr2 = matstart + opr2 or datastart + opr2 opr1 = datastart + opr1 txtadrs++
PRINT opr1	txtadrs, LD/LDI,0,0,0,opr1	LD if opr1 is VAR LDI if opr1 is NUMBER txtadrs++
	txtadrs, ST,0,0,0,0	txtadrs++
READ opr1	txtadrs, LD,0,0,0,0	txtadrs++
	txtadrs, ST,0,0,0,opr1	txtadrs++
OP,op opr1 opr2	txtadrs, LD/LDI,0,0,0,opr1	LD if opr1 is VAR or MAT LDI if opr1 is NUMBER opr1 = matstart + opr1 or datastart + opr1 txtadrs++
	txtadrs, op,0,0,0,opr2	op is ADD,SUB,MUL,DIV or their immd form if opr2 is NUMBER opr2 = matstart + opr2 or datastart + opr2 txtadrs++
	txtadrs, ST,0,0,0, matl	matl = matstart + lineno txtadrs++
END	txtadrs, HLT,0,0,0,txtstart	
LABEL opr1		vartab[opr1] = txtadrs

Table 2.2 (*Cont'd*)

Intermediate	VM1 machine code	`codegen` actions and remarks
IF opr1	txtadrs, LD,0,0,0,opr1	opr1 = matstart + opr1 or datastart + opr1 txtadrs++; push(txtadrs)
	txtadrs, JN,0,0,0,0	the 0 address will be replaced txtadrs++
GOTO opr1	txtadrs, JMP,0,0,0, gotoadrs	opr1 = vartab[opr1] gotoadrs = opr1 – txtadrs-1 txtadrs++
FI	opr1, JN,0,0,0,gotoadrs	opr1 = pop() gotoadrs = txtadrs-opr1-1 replace JN instr as a JN "here" we are overwriting

2.2.5 Comments on the Compiled Code

The compiled code for the example program was executed on the VM1 simulator described in Section 2.3 and executed properly. If you compare the "machine code" given there with the compiled code given in Section 2.2.4, you will notice considerable differences, though both of them achieve exactly the same computation. The compiled code is almost double the size of a code written by an expert VM1 programmer.

The most noticeable difference is in the use of unnecessary Load and Store instructions. If a previous instruction has left a useful value in the A register, it is not necessary to load it again. Removal of such redundant instructions is a kind of optimization, which a good real-life compiler will definitely provide.

Also, the Jump instructions can be trimmed – instead of JN … JMP … sequence to encode `if a goto loop;` statement, we could have used a single JP instruction.

In the simple example that we took up as illustration, it is not very evident, but we could improve the compiled code by utilizing the additional machine registers available and avoid using the temporary memory locations to store values.

We shall discuss these matters in more details in later chapters on "Optimization", Chapters 9 and 10.

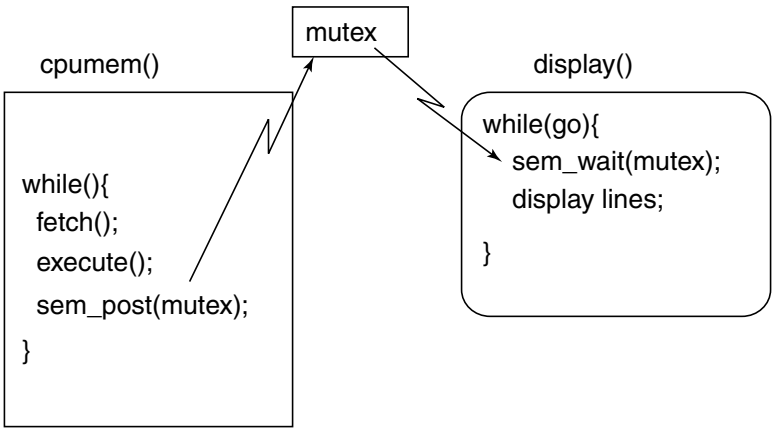
2.3 A Virtual Machine for Simple

The virtual machine VM1, given in C source code `vml.c`, executes a code given in VM1 "assembly" language form. It has a live display, which shows contents of all the machine registers and some portion of the memory, for execution of each instruction. It uses POSIX threads and `ncurses`, a CRT text screen handling package, which optimizes the operations for repeated updates. See Fig. 2.4 for a typical display.

The structure and the machine code for VM1 were already described in Section 2.1.2. The source code for the simulator, `vml.c`, is available elsewhere. The structure of the simulator is shown in Fig. 2.5.

```
                          VM1 Machine Simulator
IP:       10000 IR: ef000000 A:      2345 M :          0 BP:     0 SP: 2345
500000a   501000c     54e0001  600000d  506000d  ef000000    0        0
      0         0        2345     1234        a      2345    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
      0         0           0        0        0         0    0        0
Output:         4                                                IX1: a
                                                                 IX2: 0
                                                                 IX3: 0
```

Fig. 2.4 Typical display provided by VM1 simulator



execute() contains a delay after each execute cycle.

Fig. 2.5 Structure of VM1 simulator. Two threads are running – `cpumem` and `display`

The following are the major functions in the simulator:

fetch() get the next instruction from the memory as per IP.

operand() get the operand (and its address) specified in the instruction fetched.

execute() execute the instruction just fetched with the operand obtained, then sleep for a couple of seconds.

load() used in the `main()` to load the machine code from a file.

display() display registers and memory contents and wait till `execute()` is ready for the next display. This will be a few seconds before the next instruction.

The simulator reads in the program file, having one line per instruction and six fields containing hexadecimal values, separated by blank/s, as shown below:

```
<load addrss> <OP> <M> <RP> <R> <ADDRS>
```

and then immediately starts "executing" the program. The display changes dynamically as various values are put in the machine registers.

For our example program, the content of the machine program file is:

```
0100   15 0 0 0 000A
0101   06 0 0 0 0110
0102   06 0 0 0 0000
0103   05 0 0 0 0110
0104   12 0 0 0    1
0105   0A 0 0 0 FFFB
0106   FF 0 0 0 0100
0110   0 0 0 0 0
```

This is what an "experienced machine language programmer" would write. Compare it with the output code of the compiler.

LOOKING FORWARD

In this chapter, we have discussed a simple language and its compiler and a virtual machine VM1 which interprets the target language. Out of necessity, the language and compiler were kept very simple and many issues of real-life programming languages and their compilers were simply ignored.

The various phases of the compiler were "hand-coded" and no attempt was made to make them efficient or elegant.

For developing compilers for real-life languages, it is usual to use *compiler writing tools*. There are several of them available, but we shall gradually introduce two of them – a Scanner generator `lex` and its derivative `flex`, and a parser generator `yacc` (yet another compiler compiler) and its enhanced version `bison`. Much of the tedium of writing a parser and to some extent semantic analysis can be reduced by these tools as will be shown in the subsequent chapters. Just as the target language code is a transformed version of the source code, the compiler itself is obtained by a series of transformations from the grammar and semantic rules of the HLL. Many of these transformations are deterministic and that is how compiler writing tools came into being.

In the rest of the chapters, we discuss theory and practice of lexical analyzers, syntax analyzers, semantic analysis, enhanced grammars which aid semantic analysis, code generation options, run-time environments, etc. As we go deeper into each of these topics, it will be necessary to always keep the overall framework in mind, and it is hoped that the discussions in this chapter will help in that direction.

? EXERCISES

1. Some languages distinguish between *keywords* and *reserved words*. Suppose we want to allow a keyword as an identifier under some circumstances, how can it be done?
2. We have a variable declaration statement `int a`, available in the `simple` language. Will it not be better to have a declaration plus initialization like `int a 15` which will not only reserve space for variable 'a' but also initialize it to a value 15? What changes are required in the grammar, Scanner, parser and code generation to implement this?
3. The `if` statement has rather limited power, as no relational operators are available in `simple`. Suggest a set of relational operators to be included in `simple` and investigate what modifications are required in various phases of the compiler to implement them.
4. How would you include array variables in `simple`? Discuss how to use various addressing modes available in VM1 to implement arrays. You will need to enhance grammar, Scanner, parser and codegen also.
5. Suppose you are told to include subroutine facility in `simple` language. Investigate if you can do this with just the instruction set given. Would you like to have some additional instructions?

WEB RESOURCES

`http://www.thefreecountry.com/programming/compilerconstruction.shtml`
`http://www.fortran-2000.com/ArnaudRecipes/CompilerTricks.html`

GLOSSARY

Accumulator is a special register in the processor which takes part in almost all arithmetic and logical operations.
Acyclic graph is a graph without a cycle, i.e. a path from a vertex back to itself, possibly via a number of other vertices.
Alphanumeric characters are drawn from a set of alphabetic and numeric characters.
Backus-Naur Form See BNF
BNF is a notation for context-free grammars, often used to describe the syntax of computer programming languages, document formats, instruction sets and communication protocols. Extended Backus-Naur Form (EBNF) is extension of the basic Backus-Naur Form (BNF).
CRT – Cathode Ray Tube An older technology for graphical displays such as TV and computer video Terminals.
DAG Directed Acyclic Graph
Descending is decreasing order of some key elements in the data.
Devanagari script is used for writing several Indian languages like Sanskrit, Marathi, Hindi and also Nepali.

Encode Allocates a code to a set of data elements.
Expression is a valid arrangement of variables, constants, operators and factions.
Hexadecimal base 16 positional number system.
Indexed addressing involves adding, possibly a multiple of, an integer index value to a base address, generally used for accessing an array.
Initialize Prepare the initial state of a program's variables, data structures, etc.
Interpret a statement means finding out what action the statement implies and execute that action.
Keywords have some specific semantic significance or meaning in the source language. Keywords are usually also reserved words.
Machine-dependent optimization methods try to make the target code corresponding to the source program more efficient and less resource hungry, using the techniques utilizing particular characteristics and facilities of the architecture of the processor.
Machine-independent optimization methods try to make the target code corresponding to the source program more efficient and less resource hungry, using the techniques independent of the architecture of the processor.
Mapper is another name for semantic analyzer. It converts or maps the parse tree into an Intermediate Representation.
Meta-literal is a symbol which belongs to a meta-language and stands for a particular construct in the object language.
Meta-variable a symbol which belongs to a meta-language and stands for elements of some object language. For example, in the sentence "Let S1 and S2 be two sentences of a language L" the symbols S1 and S2 are part of the meta-language in which the statement about the object language L is written.
Optimize is the process of modifying a program to make some aspect of it work more efficiently or use fewer resources.
Parser is another name for the syntax analyzer, which identifies language elements like variables, numbers, keywords and builds the parse tree.
Parse tree is a representation of a statement of the source program in the form of a tree, showing syntactical relationships between various tokens in the statement.
Pedagogic is related to teaching or training.
POSIX – "Portable Operating System Interface for Unix" A family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utility interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.
Reserved words are words or identifiers reserved for special use by the language grammar. They must not be used as user variables in the source program.
Right-iterative form of a grammar contains production rules in which the RHS is in the form of iteration or repetition of some part of the string on the right-hand end. For example, $A \rightarrow B\ \{C\}$, where the braces indicate indefinite numbers of times that C can be added.
Right-recursive form of a grammar contains production rules in which the RHS is in the form of recursion on the right-hand end. For example, $A \rightarrow B\ A$.
Scanner is another name for a lexical analyzer. It breaks down source programs into the language atoms, creates their internal representations and prepares a Symbol Table.
Simulator is a software which mimics the actions of a hardware which is not available, not present or not being used.

Threads of control flow are allowed by most modern operating systems, allowing multiple streams of code being executed on the processor.

Two-address instructions contain at most two memory references.

Unsigned-number is a number without the sign information, always considered to be of positive value.

White spaces are characters which are not seen in a display or printout, especially *space*, *tab* and *new-line*. A compiler considers comments also as white-spaces.

3

Lexical Analyzer

What you will learn in this chapter

- The basic functions of a Scanner
- The relationship between a Scanner and the rest of the phases of a compiler
- How to implement ad hoc Scanners
- How to implement Scanners based on FSM theory
- Relationships between Regular Expressions, FSMs and Scanners
- Scanner writing tools – Lex and Flex
- Example Scanner for a C-like language
- Some initial treatment of the Symbol Table

Key Words

lexical analysis, Scanner, lex, flex

In this chapter, we discuss the first phase of a compiler – the lexical analyzer or Scanner.

Parsing is a general term meaning analyzing a sentence to arrive at its syntactic structure. From theoretical viewpoint, lexical analysis is also parsing, only the atoms of the language involved are at finer granularity.

Table 3.1 Parsing: Scanner and Parser

Item	Scanner	Parser
Language atoms	Characters	Tokens
Structure sought	Tokens for identifiers, numbers, operators	Statements, expressions, control constructs
Usual grammar	Regular	CFG
Usual acceptor	FSM	PDA

Although a typical programming language contains several context-sensitive rules, e.g. identifier types, goto labels, number of arguments in a functions, etc., there is no known parsing algorithm for context-sensitive grammars. We therefore define the programming languages in terms of regular (Chomsky type 3) and context-free (Chomsky type 2) grammars (Appendix A) and use context-sensitive rules separately while performing the semantic analysis.

We shall assume that the reader has some background in formal languages and Automata, but it is strongly suggested that the material in Appendix A be reviewed. We have defined and explained some of the terms – *sentential form*, *phrase*, *simple phrase* and *handle* – used in the discussion of parsing in Section A.5.2.

We shall see the full details of parsing in Chapter 4 on Parser.

3.1 Scanner

The job of a Scanner or lexical analyzer is identifying low-level language constructs – language atoms like identifiers, keywords, numbers, operators, string literals, etc. As these constructs can be represented as regular languages, a Scanner design is based on regular expressions and uses a finite-state machine as its implementation model.

The basic scanning process is a character-by-character examination of the input source code and identifying the tokens. A real Scanner will have to do several further jobs, first of which is to supply an internal representation of the atoms, called *tokens*, to the next phase – the parser. Consider a grammar for an arithmetic expression.

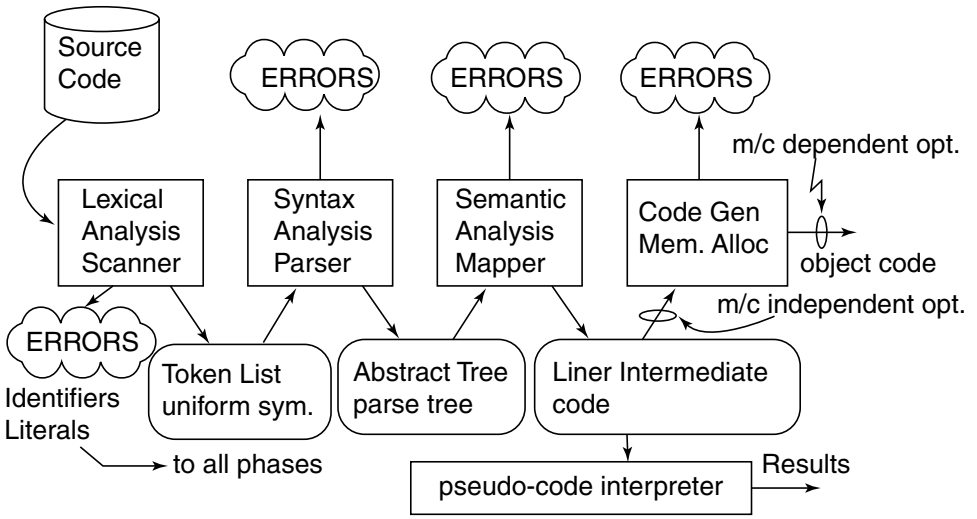


Fig. 3.1 Phases of a compiler: Scanner

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | i
```

This grammar is for syntax analysis and considers ‘i’ as a Terminal atom. To write the full grammar, we would have to write the grammar to detect the identifier atom written as ‘i’:

```
I -> I | IL | ID
L -> a | b | ... | z
D -> 0 | 1 | ... | 9
```

This is the grammar for the lexical analysis or the Scanner, which will detect the identifier atoms in the source code and supply to the parser only a token '*i*' whenever one '*I*' is detected. It will also create a *Symbol Table* of the identifiers. There are two possible implementations for a Scanner:

1. A separate pass: generates an intermediate file of uniform symbols (tokens), for the Parser.
2. As a co-routine: (or sub-routine) it is called by the parser whenever a next token is needed.

The following are the advantages of the sub-routine method:

1. The complete token list for the whole of the source code need not be stored.
2. Multiple Scanners are possible, depending upon some detected syntax construct.
3. Scanners can get feed-back from the parser to handle some difficult languages like FORTRAN. Consider two FORTRAN statements: `DO11I=1.5` and `DO11I = 1,5.` Both are valid in FORTRAN, but the Scanner will have to identify `DO 11 I` as separate atoms in the second case.

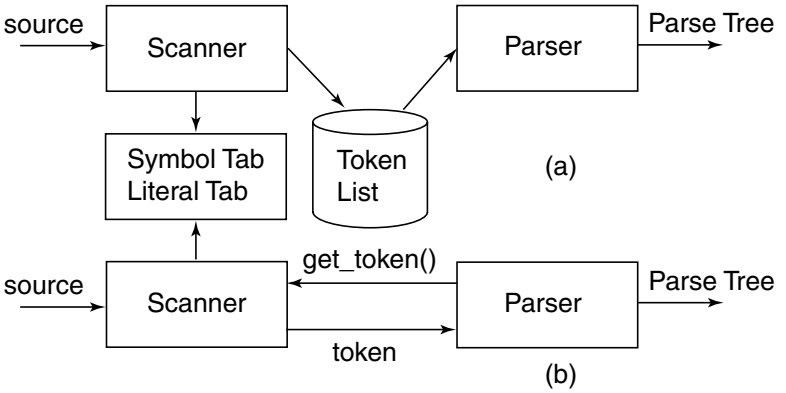


Fig. 3.2 Two ways in which a Scanner can interact with the parser: (a) Scanner creates an intermediate file of token-list; (b) Parser uses the Scanner as a procedure and calls it when a token is needed

Generally, in order to avoid the complexity introduced by the above-mentioned problem, most of the programming languages use a space character as a delimiter and restrict the use of the keywords severely.

Whenever the Scanner identifies an atom, it returns a *token* which is a pair consisting of an integer denoting the type of token and generally an index into a table where the *value* of the atom is stored. These values are usually as shown in Table 3.2.

Table 3.2 Token value component for different atoms

Atom	Example	Token value	Table	Table entry
Number	1234	Index to table	Constants	Constant value
Identifier	Voltage	Index to table	Identifiers	Identifier
String literal	"answer is"	Index to table	Strings	Pointer to string
Operator	+	Operator itself	–	–
Keyword	While	Index to table	Keywords	Keyword

The token type is usually denoted by an integer, one specific value for each syntactic type. Following the convention used by `yacc` (see Section 3.3), the token type is assigned values greater than 256. This will allow the one character operators, which are most frequent, to be denoted by the token type itself. For example, consider statements

```
label1 : a = a + 5 ; goto label2;
```

and following illustrative assignments for syntactic types:

Syntactic type	Type	Operator	Value
IDENT	257	:	0x3A
CONS	258	;	0x3B
GOTO	259	=	0x3D
		+	0x28

The Scanner will send out the following tokens to the parser.

Token type	257	0x3A	257	0x3D	257	0x28	258	0x3B	259	257	0x3B
Token value	1	0	2	0	2	0	1	0	0	3	0

A typical Scanner will have to do the following jobs:

1. Detect and report the use of illegal characters as errors in the source code.
2. Remove comments and white spaces (blanks, tabs, new lines).
3. Differentiate between identifiers and the keywords.
4. Prepare Symbol Table, Constant or Literal Table and convert the literals to internal representations (e.g. 123.45 is stored internally in IEEE standard format as 0x42f6e666).
5. Send the token to the parser.

At this stage, it is suggested that you review the material in Appendix A, especially the topics regular languages, regular expressions and finite-state Automata.

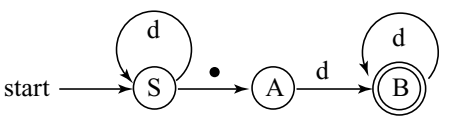
3.1.1 Examples: RE, FSM and Implementation

We now discuss several examples of regular expressions, corresponding FSMs and their implementation in the form of algorithm or code.

Example: A decimal number

We want to detect unsigned decimal numbers of the form $\underbrace{d\ldots d}_{\text{0 or more}} \cdot \underbrace{d\ldots d}_{\text{1 or more}}$.

The corresponding RE is R1 $= d * .dd*$ where d is any digit 0 to 9.
The FSM is shown below.



The corresponding Scanner code in C is

```
int main(){
   char a;
   S : a = getch();
       if((a >= '0') && (a <= '9')) goto S;
       if(a == '.') goto A;
       else {
          printf("\nError");
          goto S;
       }
   A : a = getch();
       if((a >= '0') && (a <= '9')) goto B;
       else {
          printf("\nError");
          goto S;
       }
   B : a = getch();
       if(a == '#'){
          printf("\nAccepted");
          goto S;
       }
       if((a >= '0') && (a <= '9')) goto B;
       else {
          printf("\nError");
          goto S;
       }
}
```

Note that we added an additional input character '#' to indicate the end of input string. We could have used EOF character instead. We could have used a separate function `get_d()` to check if the input character is in the range '0' to '9'. We have not included *actions* to be taken on acceptance of the input string. With a separate `get_d()` function, the implementation becomes:

```
int get_d(){
    char a;
    a = getch();
    if((a >= '0') && (a <= '9')) return 'd';
    if(a == '.') return 'p';
    if(a == '#') return 'e';
    else {
         printf("\nError");
         return 0;
    }
}
int main(){
    char a;
    S : if((a = get_d()) == 'd') goto S;
        if(a == 'p') goto A;
        else goto S;
```

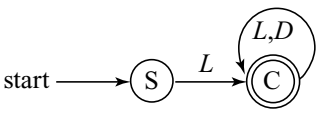
```
  A : if((a = get_d()) == 'd') goto B;
      else goto S;
  B : if(get_d() == 'e'){
         printf("\nAccepted");
         goto S;
      }
      if(get_d() == 'd') goto B;
      else goto S;
}
```

Note that `get_d()` has effectively become our Scanner's Scanner.

Example: Identifier

We want to detect identifiers defined by an RE R2 = $L(L \mid D)^*$, where L represents letters 'A to Z a to z' and D represents digits '0 to 9'.

The FSM is:

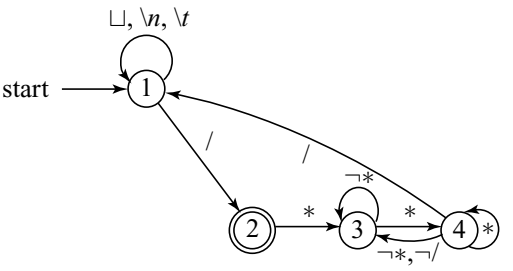



```
int main(){
    char a;
    S : a = getch();
        if(isalpha(a)) goto C;
        else {
             printf("\nError");
             goto S;
        }
    C : a = getch();
        if(isalpha(a) || isdigit(a)) goto C;
        else {
             printf("\nError");
             goto S;
        }
}
```

Example: A Scanner for C-like language

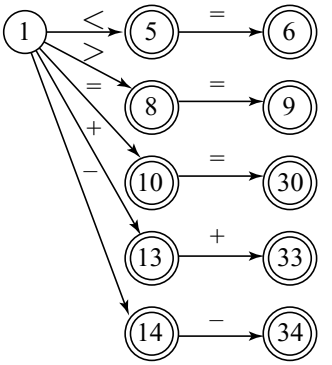
As a final example, we discuss a simplified Scanner for a C-like language. The simplification is mainly in ignoring the pre-processor statements, i.e. those starting with '#'.

One of the tricky issues is handling the comments; they are to be ignored, just like white space. So we begin by giving portion of the FSM which deals with white space and comments.



Note that the final state 2 accepts a '/', but there is a transition out of it on a '*'. Such transitions

will require *look-ahead* in our implementation. The blanks '␣', new line and tab characters are just ignored, so the transition is back to state 1. When a '/' is encountered, a look-ahead will check if the next character is a '*', if so, then we are in a comment and go to state 3. We remain in state 3 till a '*' is again encountered. Then possibly we are coming out of the comment and in anticipation of that, go to state 4. If the very next character is not a '/' or a '*' we go back to state 3, as the comment is not over yet. If it is a '*', we remain in state 4. In state 4, if we get a '/', the comment is over and we go back to state 1.



Other situations where we shall have to use look-ahead is when we encounter multi-character operators, such as '>=' or '++'. The portion of FSM corresponding to them is shown below.

The other double-character operators in C, such as '!=', '||' and '&&' will have similar state transitions.

Single-character operators such as '(', ')', ':', ';', etc. take the FSM to a state for each of them. We have already seen detection of numbers and identifiers in the previous two examples. A skeleton C language code is given below.

```
Scanner()
{
  int c;
  char *p;

  L1: while((c = getchar()) == ' ' || c == '\t' || c == '\n');

  if(c == EOF)
    return 0;

  if(isalpha(c)){
      Symbol *s;
      char sbuf[100], *p = sbuf;
    do{
      *p++ = c;
    }
    while((c = getchar()) != EOF && isalnum(c));
    ungetc(c, stdin);
    *p = '\0';

    if((s = lookup(sbuf)) == 0){
      s = install(sbuf, UNDEF, addrs);
      addrs++;
    }
    lexval.sym = s;
    return lexval.type = s->type == UNDEF ? VAR : s->type;
  }
```

```
     switch(c)
{
     int c1;
       case '/': c1 = lookahead('*', CSTART, DIV);
                 if(c1 == DIV) return DIV;
                 else {
                      L3: c1 = getchar();
                      while(c1 != '*') c1 = getchar();
                      while(c1 == '*') c1 = getchar();
                      if(c1 != '/') goto L3;
                      goto L1;
                 }
case '>':
  return lookahead('=', GE, GT);
case '<':
  return lookahead('=', LE, LT);
case '=':
  return lookahead('=', EQ, '=');
case '!':
  return lookahead('=', NE, NOT);
case '|':
  return lookahead('|', OR, '|');
case '&':
  return lookahead('&', AND, '&');
case '\n':
  lineno++;
  return '\n';
default:
  return c;
}
     }

    if(c == '.' || isdigit(c))
      {
        double d;
        ungetc(c, stdin);
        scanf("%lf", &d);
        yylval.sym = install("", NUMBER, d);
        return NUMBER;
      }
    return c;
}

int lookahead(int check, int yes, int no)
{
    int c = getchar();
    if(c == check)
    return yes;
  ungetc(c, stdin);
  return no;
}
```

3.2 Symbol Tables and a Scanner

Though as suggested above and in Figs. 3.1 and 3.2, Symbol Tables are created during the Scanner phase of a compiler, their real use and manipulations will be needed during the semantic analysis phase and as such we postpone full discussion of Symbol Table handling techniques till Section 5.5 in Chapter 5. However, the Scanner does initiate building up of the Symbol Table and here we discuss building a skeleton Symbol Table, with whatever information is available with the Scanner.

A general discussion about Symbol Tables is given in Appendix B, Section B.2.3.

In anticipation of later developments, we build the Symbol Table as a binary search tree (BST). Though a BST can have worst-case search time as bad as a linked list, we use it here for the sake of simplicity. In a real-life compiler, a more sophisticated binary tree structure like AVL or Red–Black trees or Hash tables would be used, in order to get a fast access to the Symbol Table. The design we present here is a slightly simplified version of the Symbol Table that will be built for a full example compiler in Chapter 12.

Each node in the Symbol Table BST has the following structure, declared in the file `tb_tree_l.h`:

```
typedef union{
  struct node_struct*N;
  char *S;
  long I;
  float F;
  void *P;
}util;
typedef struct node_struct{
struct node_struct *link[2]; // left = 0, right = 1
util u,v,w,x,y,z;
}Node;
```

The links to the left and right child are stored in a two-cell array `link[2]`. There are six elements of type `util`, which can be used for storing up to six attributes for each node. We shall need to use them in later chapters, where their application will be detailed. Note that though this node structure has been optimized for a binary tree, it can easily be used for a general tree.

As far as a Scanner is concerned it will create a Symbol Table entry for each identifier, constant, string literal, keywords, built-in functions, predefined identifiers and constants, etc., along with their syntactic type and value if known. The utility fields are defined as follows:

```
#define L(n)          ((n)->link[0])
#define R(n)          ((n)->link[1])
#define name(n)       ((n)->w.S)
#define type(n)       ((n)->v.I)
#define val(n)        ((n)->u.F)
```

Some of the type values are:

```
#define NUMBER 258
#define STRING 259
#define VAR 261
#define UNDEF 263
```

The Scanner that we plan to use in Chapter 12 is similar to the one we have discussed above in Section 3.1.1, and hence its code is not detailed here. The insertion and look-up of Symbol Table entries are carried out by the following functions, which in turn invoke standard BST handling functions from our Tree Data Structure library, called `tree-base`.

```
typedef Node Symbol;
static Symbol *symtab; /* Symbol Table */

void insert_node(Node **r, Node * n);
Node *search_node(Node *r, Node *n);
/*----- find s in symbol table -----*/
Symbol* lookup(char* s){
        Symbol *sp;
        sp = malloc(sizeof(Symbol));
        name(sp) = malloc(strlen(s)+1);
        strcpy(name(sp), s);
        sp = search_node(symtab, sp);
        return sp;
}
/*----- install s in symbol table -----*/
Symbol* install(char* s, int t, double d){
        Symbol *sp;
        sp = malloc(sizeof(Symbol));
        name(sp) = malloc(strlen(s)+1);
        strcpy(name(sp), s);
        type(sp) = t;
        val(sp) = (float)d;
        insert_node(&symtab, sp);
        return sp;
}
void traverse_rec(Node *r);
void traverse(){
  Node * r = symtab;
  traverse_rec(r);
}
```

In our `tree-base` library, the BST manipulation functions are written as abstract data types (ADT) and require that application-specific functions be written to handle the comparison of values and input–output from a data structure.

```
/*------ comp, display_value ------*/
int comp(Node *i, Node *j){
  return strcmp(i->w.S, j->w.S);
}
void display_value(Node *n){
  if(n == NULL) printf("No value available\n");
  else printf("[%s](%d){%x}\n", n->w.S,n->v.I,n->u.P);
}
```

Note that the `display_value()` matches our definition of the utility fields. As the value associated with a node could be of several types, we simply print it as an unsigned hex value when we want to dump the Symbol Table.

The BST insert and search functions are:

```
/*---------- insert -----------*/
void insert_node(Node **r, Node * n){
  if((*r) == NULL)
    {
      (*r) = n;
      L(*r) = NULL;
      R(*r) = NULL;
      return;                  /* tree was empty */
    }
  if(comp(n, (*r)) < 0)
    insert_node(&L (*r), n);
  else
    insert_node(&R (*r), n);
  return;
}
/*---------- search ----------*/
Node *search_node(Node *r, Node *n){
  if(r == NULL) return NULL;
  if(comp(n, r) < 0)
    return search_node(L(r), n);
  else if(comp(n, r) > 0)
    return search_node(R(r), n);
  else return r;
}
```

Finally, we would like to have a look at the Symbol Table created, for which we have a `traverse()` function.

```
/* ---------- traverse BST ---------- */
/* need to do in-order traversal      */
void traverse_rec(Node *r){
  if(r == NULL)
    return;
  traverse_rec(r->link[0]);
  display_value(r);
  traverse_rec(r->link[1]);
}
```

After writing a suitable `main()` function, which reads an input file (possibly STDIN), invokes the Scanner, and dumps the Symbol Table, we gave the following input:

```
bbb = 1
aaa = 2
ccc = 3
r = aaa + bbb + ccc
```

The output from the Scanner was:

```
261 61 258 10
261 61 258 10
261 61 258 10
261 61 261 43 261 43 261 10
```

Note that 61, 10 and 43 are decimal ASCII codes for '=', 'nl' and '+', respectively. We also obtained the following output as Symbol Table dump:

```
[1.0000000e+00](258){3f800000}
[2.0000000e+00](258){40000000}
[3.0000000e+00](258){40400000}
[aaa](263){0}
[bbb](263){0}
[ccc](263){0}
[r](263){0}
```

Note that the numbers are also stored in the Symbol Table, with their display value as "name". This is an option we have selected in our design. Also note that what we see as *variables* is stored as UNDEF (263) and not VAR (261) because the Parser has not yet identified them as validated variables. Note further the zero values for the identifiers and floating-point hex values for the numbers.

We also had another version of this Scanner, modified to print the BST. It gave the following output for the same input.

```
                    bbb(263)
                       / \
                      /   \
                     /     \
                    /       \
                   /         \
                  /           \
1.0000000e+00(258)          ccc(263)
                  \             \
            aaa(263)          r(263)
                  /
2.0000000e+00(258)
                  \
3.0000000e+00(258)
```

In the subsequent chapters, we shall have occasion to refer to the Symbol Table for actions suitable for a particular phase of the compiler.

3.3 Compiler Writing Tools

Till now we have seen a few versions of hand-coded Scanners. Their designs were more or less *ad hoc* in the sense that a bit of common-sense and some amount of cut and try were used to implement them. As mentioned previously in Chapter 2, the operations of a Scanner can be specified in terms of a regular grammar, and systematic transformations of this grammar can give us a Scanner. The

compiler writing tools such as `lex` and `yacc` use this idea to reduce the effort and time involved in implementing a compiler to considerable extent.

Before 1975 writing a compiler was a very time-consuming process. Most of the time it involved "boot-strapping" a compiler – programming in assembly language, write a hand-coded compiler for a watered-down language and then use that compiler to develop a more powerful compiler and so on. In 1975, Lesk [LS75] and Johnson [Joh75] published papers on two compiler writing tools – `lex` and `yacc`. There are enhanced versions of these tools available – `flex`, `byacc` and `bison` (improvement over `yacc`). In addition, as Perl language is very handy in manipulating text strings, Scanners and Parser generating tools are developed in Perl also – `Parser::Yapp` and `Parse::lex`. These utilities greatly simplify compiler writing and has been used and being used quite extensively for that purpose. We discuss here `lex` and `flex`, and in subsequent chapters `yacc`.

3.3.1 Lex – A Scanner Generator

You code regular patterns and feed them to `lex`. It will read your patterns and generate C code for a lexical analyzer or Scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. A token usually consists of a pair (type, value). This is illustrated in Fig. 3.3.

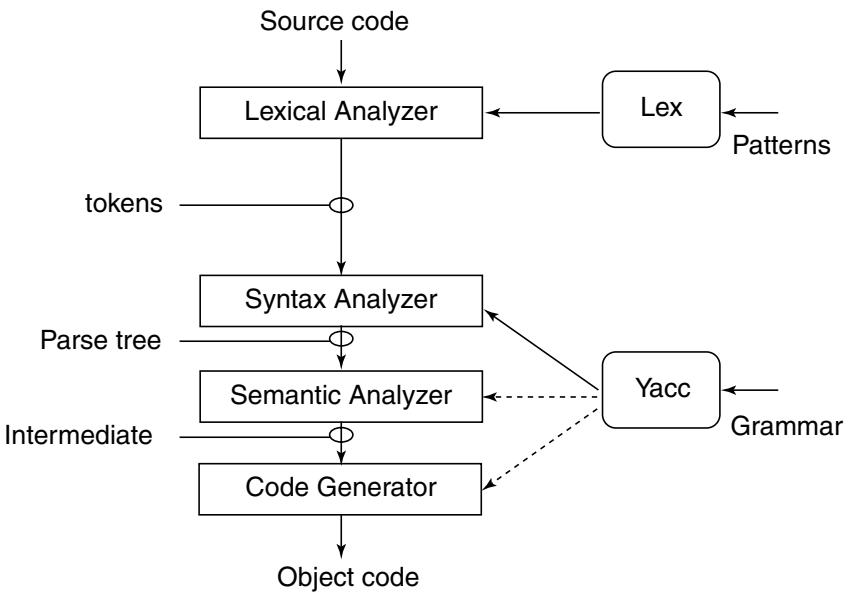


Fig. 3.3 Compilation using translator writing tools `lex` and `yacc`. Yacc generates a full parser and makes provision for including *Actions* for semantic analysis and code generation, in the form of program hooks. The dotted lines show this relationship

When the lexical analyzer finds identifiers in the input stream it enters them in a Symbol Table. The Symbol Table may also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate Symbol Table index.

You code a grammar and feed it to `yacc`, which will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure of the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next phase of the compiler does semantic analysis and generates an intermediate code. The next step, code generation, converts this intermediate code into an object code. Some compilers produce machine code, whereas others output an assembly language code to be processed by an assembler.

Figure 3.4 illustrates the file naming conventions used by `lex` and `yacc`. Suppose our goal is to write a compiler for the language called `lang`. First, we need to specify all pattern-matching rules for lex (`lang.l`) and grammar rules for `yacc` (`lang.y`). Commands to create our compiler, `lang`, are listed below:

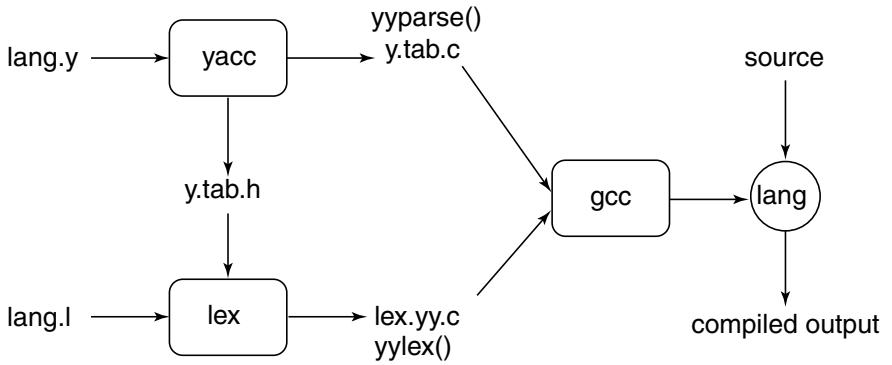


Fig. 3.4 Building a compiler using `lex` and `yacc`. File naming conventions. For `flex` and `bison`, these conventions are relaxed

```
yacc -d lang.y                  # create y.tab.h, y.tab.c
lex lang.l                      # create lex.yy.c
cc lex.yy.c y.tab.c o lang      # compile/link
```

Yacc reads the grammar descriptions in `lang.y` and generates a syntax analyzer (parser) that includes a function `yyparse()`, in file `y.tab.c`. Included in file `lang.y` are token declarations. The `-d` option causes `yacc` to generate definitions for tokens and place them in file `y.tab.h`. `lex` reads the pattern descriptions in `lang.l`, includes file `y.tab.h` and generates a lexical analyzer that includes function `yylex()`, in file `lex.yy.c`. Finally, the Scanner and parser are compiled and linked together to form the executable `lang`.

From `main()` in `y.tab.c`, `yyparse()` is called to run the compiler. Function `yyparse()` calls `yylex()` to obtain each token, whenever it needs one.

Note that `lex` uses regular expressions (REs) as specifications of the strings you want to detect and not a regular grammar (RG). This is so because a RE is more compact than corresponding RG and easier and more intuitive to understand.

We have seen that a FSM can be used as a blue-print for a C language program which mimics the actions of the FSM. This very same technique is used by `lex`. Regular expressions are translated by `lex` to a computer program that mimics an FSM. Using the next input character and current state, the next state is easily determined by indexing into a computer-generated state table. Now we can easily

understand some of `lex`'s limitations. For example, it cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a '(', we push it on the stack. When a ')' is encountered, we match it with the top of the stack and pop the stack. However, `lex` has only states and transitions between states. As it has no stack, it is not well suited for parsing nested structures. Yacc augments an FSM with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at RE pattern matching.

A number of extensions to basic RE operations of concatenation, alternation and repetition are used in `lex` as given in Table 3.3.

Table 3.3 Pattern-matching primitives

Meta-character	Matches
.	Any character except new line
n	New line
*	Zero or more copies of the preceding expression
+	One or more copies of the preceding expression
?	Zero or one copy of the preceding expression
x{*m*, *n*}	*x* repeated *m* to *n* times (inclusive)
^	Beginning of line, also "not" in a character class
$	End of line
a | *b*	*a* or *b*
(*ab*)+	One or more copies of *ab* (grouping)
"*a*+*b*"	Literal "*a*+*b*" (C escapes still work)
[0 − 9]	Character class
ab/*cd*	Match *ab*, but when followed by *cd*
x	Escape *x*, matches literal *x*
{*varname*}	Substitute a predefined sub-pattern

Some pattern-matching examples are shown in Table 3.4. Note that within a character class, normal operators lose their meaning.

We say that `lex` uses *greedy* matching. If two strings match the same RE, the longest match wins. In case both matches are of same length, then the first RE pattern listed is used.

Input to Lex is divided into three sections, with %% dividing the sections.

```
definitions
%%
rules
%%
user's sub-routines written in C
```

This is best illustrated by an example. The first example is the shortest possible lex file:

```
%%
```

Table 3.4 Lex: pattern-matching examples

Expression	Matches
abc	*abc*
*abc**	*ab abc abcc abccc …*
abc+	*abc abcc abccc …*
a(*bc*)+	*abc abcbc abcbcbc …*
a(*bc*)?	*a abc*
[*abc*]	One of: *a*, *b*, *c*
[*a* − *z*]	Any one letter, *a* to *z*
[*a*\ − *z*]	One of: *a*, −, *z*
[−*az*]	One of: −, *a*, *z*
[*A* − *Za* − *z*0 − 9]+	One or more alphanumeric characters
[*t**n*] +	White space
[^ *ab*]	Anything except: *a*, *b*
[*a* ^ *b*]	One of: *a*, ^, *b*
[*a* | *b*]	One of: *a*, |, *b*
a| *b*	One of: *a*, *b*

Input is copied to output, one character at a time. The first %% is always required, as there must always be a rules section. However, if we do not specify any rules, then the default action is to match everything and copy it to output. Defaults for input and output are `stdin` and `stdout`, respectively. Here is the same example, with defaults explicitly coded:

```
%%
       /* match everything except newline */
.       ECHO;
       /* match newline */
\n     ECHO;
%%
int yywrap(void) {
      return 1;
}
int main(void) {
      yylex();
      return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by white space (space, tab or new line) and an optional *action* associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces.

Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behaviour to specify the comments in our `lex` file. In this example, there are two patterns, "." and "*n*", with an ECHO action associated with each pattern. Several macros and variables are predefined by `lex`, ECHO being one of them and it writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable `yytext` is a pointer to the matched string (NULL-terminated), and `yyleng` is the length of the matched string. Variable `yyout` is the output file and defaults to `stdout`. Function `yywrap()` is called by `lex` when input is exhausted. Return 1 if you are done, or 0 if more processing is required.

Every C program requires a `main()` function. In this case, we simply call `yylex()`, the main entry point for `lex`. Some implementations of `lex` include copies of `main()` and `yywrap()` in a library, eliminating the need to code them explicitly. This is why our first example, the shortest `lex` program, functioned properly. The predefined variables and functions are given in Table 3.5.

Table 3.5 Lex predefined variables and functions

Name	Function
`int yylex(void)`	Call to invoke lexer, returns token
`char *yytext`	Pointer to matched string
`yyleng`	Length of matched string
`yylval`	Value associated with token
`int yywrap(void)`	Wrapup, return 1 if done, 0 if not done
`FILE *yyout`	Output file
`FILE *yyin`	Input file
`INITIAL`	Initial start condition
`BEGIN`	Condition switch start condition
`ECHO`	Write matched string

Here is a program that does nothing at all. All input is matched, but no action is associated with any pattern, so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate `yylineno`. The input file for `lex` is `yyin` and defaults to `stdin`.

```
%{
     int yylineno;
%}
%%
^(.*)\n       printf("%4d\t%s", ++yylineno, yytext);
%%
```

```
int main(int argc, char *argv[]) {
     yyin = fopen(argv[1], "r");
     yylex();
     fclose(yyin);
}
```

The definitions section is composed of substitutions, code and start states. Code in the definitions section is simply copied as is to the top of the generated C file, and must be bracketed with ”%{” and ”%}” markers. Substitutions simplify pattern-matching rules. For example, we define digits and letters:

```
digit          [0-9]
letter         [A-Za-z]
%{
     int count;
%}
%%
     /* match identifier */
{letter}({letter}|{digit})*                 count++;
%%
int main(void) {
     yylex();
     printf("number of identifiers = %d\n", count);
     return 0;
}
```

White space must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces ({*letter*}) to distinguish them from literals. When we have a match in the rules section, the associated C code is executed. Here is a Scanner that counts the number of characters, words and lines in a file (similar to Unix/Linux `wc`):

```
%{
     int nchar, nword, nline;
%}
%%
\n             { nline++; nchar++; }
[^ \t\n]+ { nword++, nchar += yyleng; }
.              { nchar++; }
%%
int main(void) {
     yylex();
     printf("%d\t%d\t%d\n", nchar, nword, nline);
     return 0;
}
```

These were rather exotic uses of `lex`, but we discuss now a `lex` input file for a typical Scanner for a programming language.

```
%{
#define VARIABLE   257
#define INTEGER    258
```

```
#define TEXT       259
%}
comment            "//".*
delimiter          [ \t\n]
whitespace         {delimiter}+
uc                 [A-Z]
lc                 [a-z]
letter             {uc}|{lc}
ascii              [^\"\n]
digit              [0-9]
variable           {lc}({lc}|{digit})*
integer            {digit}+
text               \"({ascii})*\"
%%
{whitespace}       {}
{comment}          {}
{variable}         { mkname() ;
                   return VARIABLE; }
{integer}          {mkval();  return INTEGER;}
{text}             {mktext(); return TEXT;}
%%
int mkname(){
/* save variable name in a Symbol Table */
}
int mkval(){
  calculate numerical value and sane in value table */
}
int mktext(){
  save string in the string Literal Table */
}

int main(){
  int i;
  while(i = yylex())
    printf("%d", i);
    /* Here parser will take over instead of printf() */
}

int yywrap(){
  return 1;
}
```

3.3.2 Flex

Flex is a drop-in replacement for `lex`, freely available from: `http://flex.sourceforge.net`.

Flex is a fast lexical analyzer generator. It is frequently used with the free Bison parser generator. Flex was originally written in C by Vern Paxson around 1987. A similar lexical Scanner for C++ is `flex++`, which is included as part of the flex package. Flex is a non-GNU project, but the GNU project developed the manual for flex.

The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a Scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates a C source file named, `lex.yy.c`, which defines the function `yylex()`. The file `lex.yy.c` can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

If you are using Linux as your program development environment, type `info flex` to see the on-line manual for `flex`.

3.3.3 Debugging `lex` and `flex`

Lex has facilities that enable debugging. This feature may vary with different versions of lex, so you should consult documentation of the version that you use for further details. The code generated by lex in file lex.yy.c includes debugging statements that are enabled by specifying command-line option `-d`. Debug output in `flex` may be toggled on and off by setting `yy_flex.debug`. Output includes the rule applied and corresponding matched text. If you are running `flex` and `yacc` together, specify the following in your `yacc` input file:

```
extern int yy_flex_debug;
int main(void) {
   yy_flex_debug = 1;
   yyparse();
}
```

Alternatively, you may write your own debug code by defining functions that display information for the token value and each variant of the `yylval` union. This is illustrated in the following example. When `DEBUG` is defined, the debug functions take effect, and a trace of tokens and associated values is displayed.

```
   %union {
      int ivalue;
      ...
   };
   %{
#ifdef DEBUG
   int dbgToken(int tok, char *s) {
      printf("token %s\n", s);
      return tok;
   }
   int dbgTokenIvalue(int tok, char *s) {
      printf("token %s (%d)\n", s, yylval.ivalue);
      return tok;
   }
   #define RETURN(x) return dbgToken(x, #x)
   #define RETURN_ivalue(x) return dbgTokenIvalue(x, #x)
#else
   #define RETURN(x) return(x)
   #define RETURN_ivalue(x) return(x)
   #endif
```

```
%}
%%
[0-9]+          {
                    yylval.ivalue = atoi(yytext);
                    RETURN_ivalue(INTEGER);
                }
"if"            RETURN(IF);
"else"          RETURN(ELSE);
```

3.4 Error Handling in a Scanner

Generally, all errors are passed on to the parser. Usually the Scanner does not print anything. Errors are communicated to the parser by returning a special error token called ERROR. Note that you should ignore the token called `error` (in lowercase), used by the parser. There are several requirements for reporting and recovering from lexical errors:

- When an invalid character (one that cannot begin any token) is encountered, a string containing just that character is returned as the error string. Resume scanning at the following character.
- If a string contains a UN-escaped new line, that error is reported as "Unterminated string constant" and scanning is resumed at the beginning of the next line – we assume that the programmer simply forgot the close quote.
- When a string is too long, report the error as "String constant too long" in the error string in the ERROR token. If the string contains invalid characters (i.e. the null character), report this as "String contains null character". In either case, scanning is resumed after the end of the string. The end of the string is defined as either
  1. the beginning of the next line if a UN-escaped new line occurs after these errors are encountered or
  2. after the closing " otherwise.
- If a comment remains open when EOF is encountered, report this error with the message "EOF in comment". The comment's contents are not to be tokenized simply because the terminator is missing. Similarly for strings, if an EOF is encountered before the close quote, this error is reported as "EOF in string constant".

We shall now see the basic steps to include errors detection by the Scanner. We shall add error detection to a typical Scanner for a programming language given in Section 3.3.1.

```
%{
#define VARIABLE 257
#define INTEGER 258
#define TEXT 259
#define ERROR 511
%}
comment "//".*
...   ...   ...
text \"({ascii})*\"
%%
{whitespace} {}
...   ...   ...
```

```
{text} {mktext(); return TEXT;}
. {return ERROR;}
%%
int main(){
int i;
while(i= yylex())
  if(i == 511){
    printf("Error! %s\n", yytext);
  } else {
    printf("%d\n", i);
    /* Here parser will take over instead of printf() */
  }
}
```

If you generate the Scanner for the above lex-code and compile and execute the resultant Scanner, it will give the following response for a valid *integer*, a valid *variable*, a valid *string*, an invalid variable and an invalid integer (or variable), respectively:

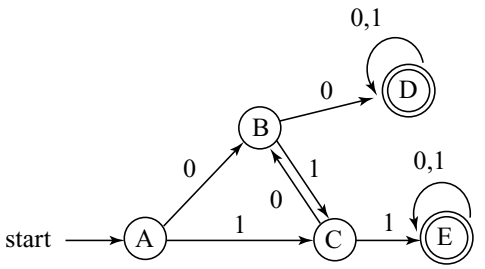
```
123
258
asd
257
"this is good."
259
ASD
Error! A
Error! S
Error! D
456wer
258
257
```

The last trial shows that though from a typical programming language viewpoint, the input string "456wer" is neither an integer nor a variable; our Scanner has detected it as an integer immediately followed by a variable. From the viewpoint of syntax (i.e. the Parser), this is a wrong construct and it should be detected as such by the parser. On the other hand, when the Scanner detects an error, normally it will resume with the next character, but the Parser will have to recover at its own recovery point. This is the reason why we said that the Scanner detected errors should be passed on to the Parser and be reported by the Parser.

? EXERCISES

1. Why do we use separate Scanner and parser in a compiler? Can the parser not do both the jobs?
2. Why in the Scanner shown in Section 3.2 the symbols like '=' do not appear in the Symbol Table?
3. Implement a `lex` based Scanner for the `simple` language discussed in Chapter 2. Compare it against the hand-coded C language implementation given there, in terms of the size of the programs and ease of developing the Scanner.

4. Implement `lex` based Scanners for all the examples for regular expressions given in this chapter.
5. If you are familiar with Perl or Python language, try to implement the Scanner for the `simple` language given in Chapter 2 in that language. Compare with the hand-coded C Scanner.
6. Can you implement a compiler for a programming language using only `lex` as the tool? What problems, if any, would you face? Hint: Look up the theory of Pumping Lemma for FSM and CFG in Appendix A.
7. Implement for C the error handling as suggested in Section 3.4.
8. Write regular expressions for the following languages over the alphabet $\sum = \{0,1\}$:
   (a) All strings that do not end with 00.
   (b) All strings that contain an even number of 1s.
   (c) All strings which do not contain the substring 10.
9. Draw DFAs for each of the languages from the above exercise.
10. Consider the following deterministic finite automaton (DFA) over the alphabet $\sum = \{0,1\}$.



Give a one-sentence description of the language recognized by the DFA. Write a regular expression for this language.

11. For each of the following specifications written in lex/Flex, give a regular expression describing the language of possible outputs. Assume that all inputs are strings consisting of characters *a*, *b*, *A* and *B*.
    (a) Specification 1:

```
[aA]                { printf("x"); }
[aA] [aA]           { printf("y"); }
[bB]                { printf("z"); }
```

    (b) Specification 2:

```
([aA]+)([bB}+)      { printf("a"); }
[aA]                { printf("b"); }
[bB]                { printf("c"); }
```

WEB RESOURCES

dragonbook.Stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis.pdf
en.wikipedia.org/wiki/Lexical_analysis
www.cs.vassar.edu/~cs331/lectures/lexical-analysis.pdf
www.mec.ac.in/resources/notes/notes/compiler/Modulel/lex.html

GLOSSARY

Bison GNU Project parser generator, a `yacc` replacement.
CFG Context-free grammar, most of the present day programming languages can be specified in terms of such a grammar.
DFA Deterministic finite-state automaton, see Appendix A.
Flex Fast lexical analyzer generator, a `lex` replacement.
GCC GNU compiler collection, developed by GNU project as a free compiler for GNU system. Front ends: C, C++, Objective-C, Fortran, Java, Ada and recently some other languages.
Generator Any entity that produces an effect or is responsible for events or results; specifically a software device to produce a sequence of output obeying certain specifications.
Intermediate Between the front- and back-ends of a compiler.
Interpreter A program that translates and executes source language statements one line at a time.
Lex A lexical analyzer generator, originating on Unix O/S.
Lexical Related to words, basic language elements.
Parse Analyze syntactically by assigning a structure making up a sentence (statement).
Parser A computer program that divides code up into functional components, i.e. does parsing.
Patterns Convention, normal, rule and formula, specifically the arrangements of characters according to specified rules.
Python An interpreted, interactive and object-oriented programming language.
Regular Expression Conforming to a standard or pattern, specifically a way of specifying simple and least complex languages and a way of specifying Chomsky Level 3 languages.
Regular Grammar Grammar which generates a regular language.
Regular Language A simple and least complex language, Chomsky Level 3 language.
Scanner A lexical analyzer; it systematically scans the input from one end to another and identifies "words" or basic language elements.
Semantic Related to the meaning of a group of sentences.
Syntax A systematic orderly arrangement, sentence structure and the grammatical arrangement of words in sentences.
Token Something serving as a representative of something else; specifically internal representation, usually an integer, for a particular language element.
Tree A data structure; examples of uses are a parse tree, syntax tree and Abstract Syntax Tree (AST).
Yacc "Yet another compiler compiler", a compiler writing tool first used in Unix O/S.
Actions The technical meaning is: the action taken when a particular syntactic construct is detected, effectively implements the "meaning" of the construct.
Associativity A property of a binary operator, which says that within an expression containing two or more occurrences of the same operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed.
Concatenation The linking together of a consecutive series of symbols.
gcc GNU C compiler.
Keyword A significant word used in indexing or cataloging, but in programming languages it means that a word which has a particular well-defined role in the syntax of the language; the keywords are generally *reserved* words.
Label An identifier used for specifying a particular sentence in a program.
Lexer See Scanner.

Lookahead Taking a parsing decision for the current input symbol, based on the next symbol which will be subsequently scanned.
Match A positive result of comparison of the input string and a pattern.
Precedence The order of operations is a rule used to unambiguously specify which operations should be performed first in a given mathematical expression.
Recovery Arriving at a well-defined and valid state, after an error is detected while performing lexical or syntax analysis; the idea is that it should be possible, after the recovery, that as much as of the rest of the code can be analyzed.

4

Syntax Analyzer

What you will learn in this chapter

- Some basic parsing terms
- Recursive-descent Parser
- Top-down parsing
- Bottom-up parsing
- Various types of LL(1) grammars and parsers
- Shift/Reduce and Operator precedence parser
- LR(0), SLR(1), LR(1) and LALR(1) parsers
- What are conflict situations in LR parsers?
- How conflicts are resolved?
- Compiler writing tools – `yacc`, `bison` and `ANTLR`

Key Words

syntax analysis, parser, RDP, Top-down, Bottom-up, LL(1), LR(0), SLAR(1), LR(1), LALR(1), conflicts, resolution of conflicts, yacc, bison

Parsing is a general term meaning analyzing a sentence to arrive at its syntactic structure. From theoretical viewpoint, lexical analysis is also parsing, only the atoms of the language involved are at finer granularity.

Though a typical programming language contains several context-sensitive rules, e.g. identifier types, goto labels, number of arguments in a function, etc., there is no known parsing algorithm for context-sensitive grammars. We therefore define the programming languages in terms of regular

Table 4.1 Parsing: Scanner and Parser

Item	Scanner	Parser
Language atoms	Characters	Tokens
Structure sought	Tokens for identifiers, numbers, operators	Statement expressions, control constructs
Usual grammar	Regular	CFG
Usual acceptor	FSM	PDA

(Chomsky type 3) and context-free (Chomsky type 2) grammars (Appendix A) and use the context-sensitive rules separately while performing the semantic analysis.

We shall assume that the reader has some background in formal languages and Automata, but it is strongly suggested that the material given in Appendix A be reviewed. We have defined and explained some of the terms – such as *sentential form*, *phrase*, *simple phrase* and *handle* – used in the discussion of parsing in Section A.5.2. Also, the definitions of an *ambiguous sentence* and an *ambiguous grammar* given there should be referred to.

4.1 Top-down and Bottom-up Parsing

Parsing of a given sentence can be done in one of the two ways shown in Fig. 4.1:

Top-down: Starting with the start symbol of the grammar, we apply various productions to ultimately obtain the required sentence.

Bottom-up: Starting with a handle in the given sentence, we construct a forest of trees by applying the reductions (i.e. productions in reverse) which will ultimately merge into one parse tree.

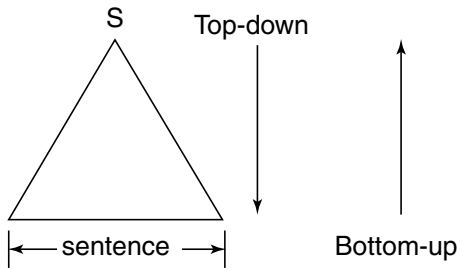


Fig. 4.1 Top-down and Bottom-up parsing

Consider a grammar:

```
G = <{I,D,L}, {a-z, 0-9}, I, P>, with productions
P = { I -> L | IL | ID
      L -> a | b | ... | z
      D -> 0 | 1 | ... | 9
    }
```

and a sentence $x2$ to be parsed. First, we consider the Top-down parsing (Fig. 4.2).

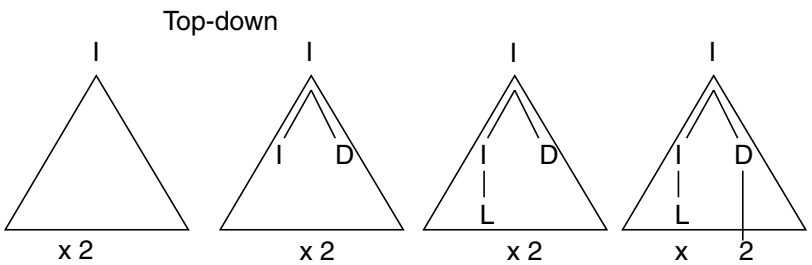


Fig. 4.2 Starting from *I*, and looking at "*x*2", we derive a single tree, which ultimately results in "*x*2"

Note that in the second step we chose the production I –> ID. Instead, had we chosen I –> IL there, the parsing would have been unsuccessful. Thus in top-down parsing, we start with the start symbol and go on applying productions, but always "looking" at the sentence we want to generate.

Now consider Bottom-up parsing (Fig. 4.3). Again take $x2$ as the given sentence to be parsed. The sequence of reductions is

$$x2 \Rightarrow L2 \Rightarrow I2 \Rightarrow ID \Rightarrow I$$

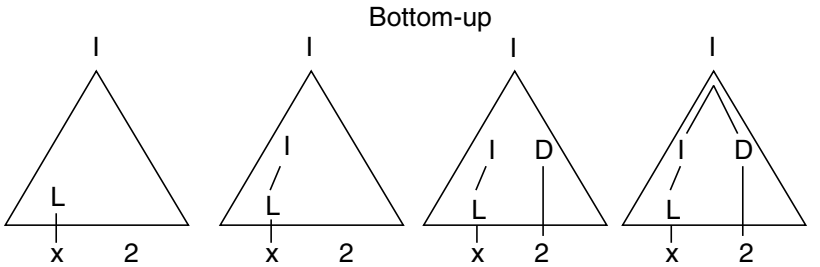


Fig. 4.3 Starting from the given sentence, we detect the handle and reduce it by grammar rules

Consider, for example, grammar G given above the sentential form LDL:

- D is NOT a phrase.
- b is a simple phrase.
- L is a simple phrase.
- L is a handle, as it is the leftmost simple phrase.

The concept of a *handle* is important as it allows us to reduce it to an NT of that grammar.

We say that during Top-down parsing, we *derive* sentential forms and during Bottom-up parsing we *reduce* from sentential forms to the start symbol. It is necessary to give formal definitions of these concepts.

Definition 4.1.1 (Leftmost canonical Direct Derivation) *If $A \rightarrow \beta$ is a production in the grammar and $\phi_1 \in V_T^*$, then $\phi_1 A \phi_2 \underset{L}{\Rightarrow} \phi_1 \beta \phi_2$.*

We say if $x \overset{+}{\underset{L}{\Rightarrow}} y$, x left produces y.

Definition 4.1.2 (Rightmost canonical Direct Derivation) *If $A \rightarrow \beta$ is a production in the grammar and $\phi_2 \in V_T^*$, then $\phi_1 A \phi_2 \underset{R}{\Rightarrow} \phi_1 \beta \phi_2$.*

We say if $x \overset{+}{\underset{R}{\Rightarrow}} y$, x right produces y.

Leftmost derivation corresponds to left-to-right scan for Top-down parsing. Rightmost reduction (i.e. derivation in reverse) is used for Bottom-up parsing, while doing left-to-right scan.

Example – Derivation and Reduction

A unambiguous version for an arithmetic expression grammar, $G_1' : V_N = \{E, T, F\}$, $\Sigma = \{+, -, *, /, (,), a\}$, $S = E$, and P is given by

$P = \{$
1. $E \rightarrow E + T \mid E - T \mid T$
2. $T \rightarrow T * F \mid T/F \mid F$
3. $F \rightarrow (E) \mid a$

$\}$

Consider the grammar G_1' and derive sentence "$a + a * a$" by left and right derivations.

Note that leftmost derivation corresponds to left-to-right scan while doing Top-down parsing. In case of Bottom-up parse, the problem of finding the correct sequence of productions $S \overset{*}{\Rightarrow} \sigma$ for a given σ is approached by starting with σ and *reducing* the string to S; see the third column of Table 4.2.

Table 4.2 Left and right derivations, right reduction

Leftmost derivation	Rightmost derivation	Rightmost reduction
$E \underset{L}{\Rightarrow} E+T$	$E \underset{R}{\Rightarrow} E+T$	$a+a*a \Leftarrow \underline{F}+a*a$
$\underset{L}{\Rightarrow} T+T$	$\underset{R}{\Rightarrow} E+T*F$	$\Leftarrow \underline{T}+a*a$
$\underset{L}{\Rightarrow} F+T$	$\underset{R}{\Rightarrow} E+T*a$	$\Leftarrow E+\underline{a}*a$
$\underset{L}{\Rightarrow} a+T$	$\underset{R}{\Rightarrow} E+F*a$	$\Leftarrow E+\underline{F}*a$
$\underset{L}{\Rightarrow} a+T*F$	$\underset{R}{\Rightarrow} E+a*a$	$\Leftarrow E+T*\underline{a}$
$\underset{L}{\Rightarrow} a+F*F$	$\underset{R}{\Rightarrow} T+a*a$	$\Leftarrow E+\underline{T*F}$
$\underset{L}{\Rightarrow} a+a*F$	$\underset{R}{\Rightarrow} F+a*a$	$\Leftarrow \underline{E}+\underline{T}$
$\underset{L}{\Rightarrow} a+a*a$	$\underset{R}{\Rightarrow} a+a*a$	$\Leftarrow E$

Note: The third column shows left-to-right scan to catch a *handle* by rightmost derivation in reverse order ($\Leftarrow$) as reduction.

Further note that for the sentential form "$E + T * a$", neither "$E + T$" nor "T" is a handle. Remember that a handle in a sentential form derived from the start symbol should be replaceable by a single NT. If we try that while reducing "$E + T * a$" as "$E * a$" it is not a sentential form derivable from E. Similarly, if the reduction as "$E + E * a$" it is not a sentential form derivable from E.

Reduced Grammars

There may be several forms of redundancies in a grammar. In a reduced grammar, which is free from such redundancies, the following are removed:

1. redundant rules like $A \rightarrow A$,
2. redundant NT, which does not generate any Terminal string,
3. unreachable NT.

4.2 Top-down Parsing

We shall now discuss some Top-down parsing in some detail. A sub-class of CFG, known as LL(k) grammars, can be parsed deterministically, i.e. without backtracking, by LL(k) methods. The nomenclature LL(k) has the following significance:

$$\overbrace{Left\text{-}to\text{-}right\ scan}^{L}\ \overbrace{Leftmost\ derivation}^{L}\ \overbrace{Number\ of\ look\text{-}ahead\ symbols}^{(k)}$$

We cannot have a grammar with left recursion for Top-down parsing. Consider a production

```
T -> T * a
```

Because Top-down parsing involves leftmost derivation, we shall always replace the T on the RHS by "$T * a$" and thus we shall have an infinite loop. We shall have to rewrite any left-recursive grammar as either right recursive or right iterative. How do we do this conversion? The general rules are as follows:

$$\textit{left recursion to EBNF: } A \rightarrow Aa \mid \beta \Rightarrow A \rightarrow \beta\{a\}$$
$$\textit{right recursion to EBNF: } A \rightarrow aA \mid \beta \Rightarrow A \rightarrow \{a\}\beta$$
$$\textit{common sub-expr: } A \rightarrow a\beta \mid a \Rightarrow A \rightarrow a[\beta]$$

Here, $\beta \in (V_N \cup V_T)^*$ not starting with A.

Within the class LL(k) there are sub-classes, with certain restrictions on the kind of production rules allowed, which will allow us to use simpler methods for parsing. LL(k) has the following most used sub-classes:

1. Recursive-descent parser (RDP),
2. Simple LL(1),
3. LL(1) without ϵ-rules,
4. LL(1) with ϵ-rules,

with progressively more complex grammar rules.

4.2.1 Recursive-descent Parser (RDP)

RDP has no left-recursive rule, either right-recursive or iterative rules (extended BNF) only. The extended BNF can be thought of as a set of rules whose RHS are generalized form of regular expressions. We have already encountered such parsers in Chapters 1 and 2.

Generally, we require a recognizer more powerful than FSM, a push-down automata (PDA) (an FSM with a separate stack) to parse sentences generated by a CFG (Fig. 4.4).

In case of RDP, an explicit stack is not present, rather we use the return-stack normally available as a hidden stack in an HLL like C (or even in most of the modern CPU hardware). Thus, the information

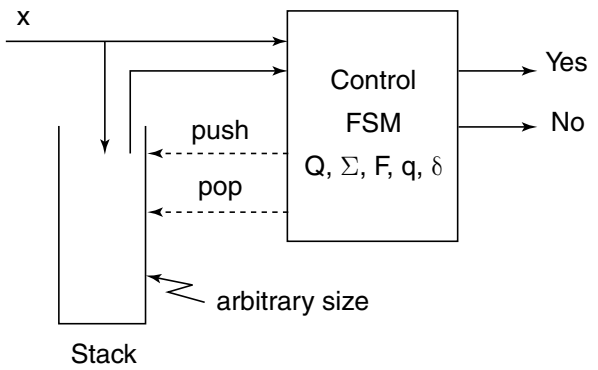


Fig. 4.4 A PDA consists of an FSM and a Stack

that would have been stored normally on the stack of the PDA gets stored on this hidden stack, which stores the Activation Record (see Section 7.1.3 in Chapter 7). The local variables within a function and function called arguments are part of the activation record and thus automatically get stacked and unstacked on the hidden stack.

There is a function for each symbol $X \in V_T \cup V_N$. This function is named `pX()`, designed to recognize any string derivable from X.

If $X \in V_T$: `pX()` simply reads the input symbol and checks that it is really X, otherwise raises an error.

If $X \in V_N$: within `pX()` we take action depending upon the next (look-ahead) symbol and call `pX1()`, `pX2()`, etc. depending upon its value.

Example

Consider the arithmetic expression grammar G_1' given previously. We have to rewrite it in an extended BNF, which is a right-iterative form, using the technique given previously as follows:

```
E -> T { + T }
T -> F { * F }
F -> ( E ) | a
```

A skeleton code for the corresponding RDP is:

```
int nextsym(){ /* look-ahead next input symbol */}
int symbol(){ /* get next input symbol */}
int error(){/* report error and exit */}

pE(){ pT(); while(nextsym() == '+'){pplus(); pT();} }
pT(){ pF(); while(nextsym() == '*'){past(); pF();} }
pF(){ switch(nextsym())
        case '(': plp(); pE(); prp(); break;
        case 'a': pa();
      }
pa(){ if(symbol() != 'a'} error(); }
plp(){ if(symbol() != '('} error(); }
prp(){ if(symbol() != ')'} error(); }
pplus(){ if(symbol() != '+'} error(); }
past(){ if(symbol() != '*'} error(); }
```

RDP vs. A Parser Generator

We have seen that many real-life compiler writing projects use a parser generator like `yacc`. Looking at the simplicity of an RDP, how does development of a compiler using RDP compare with development using a parser generator?

- Probably more work involved in developing an RDP.
- Less risk of a disaster – you can almost always make a recursive-descent parser work.
- May have easier time dealing with resulting code: you need not learn the meta-language of the parser generator, no need to deal with potentially buggy parser generator, no integration issues with automatically generated code.
- If your parser development time is small compared to the rest of project, or you have a really complicated language, use hand-coded recursive-descent parser.

4.2.2 Exercises

1. Implement the RDP fully for arithmetic expression grammar, including '−' and '/' operations also. Your implementation should display how the parsing is being done by showing the steps taken. You may have to add an extra initial production like `S -> E #` and corresponding function, for ease of handling the input string.
2. We have simply terminated the RDP on an error detection. Implement error handling at three levels – detection, report position and recovery from error.

4.2.3 Parser for Simple LL(1) Grammars

A simple LL(1) grammar is a CFG without ϵ-rules and have rules of the form

$$A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid a_3\alpha_3 \mid \ldots$$

where $a_i \in V_T$ and $\alpha_i \in (V_T \cup V_N)^*$, $A \in V_N$. That means, for a particular NT, each alternative rule starts with different Terminal symbols. (In fact, BASIC was designed like this.)

Example

Given a grammar with productions:

```
P = { 0. S -> E#
      1. E -> a
      2. E -> (E)
      3. E -> bA
      4. A -> +E
      5. A -> *E
}
```

Consider an input string "$b + b * a\#$".

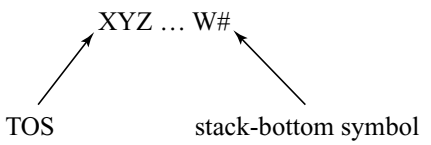
We assume an explicit stack in the PDA. The PDA stacks the *unmatched* portion of the sentential form, while developing the syntax tree (Fig. 4.5).

We use a *parsing function*, which is the specification of the FSM, in the form of a table:

$$M : \{V \cup \#\} \times \{V_T \cup \#\} \rightarrow \{(\beta, i),\ \ pop, accept, error\}$$

$$M(X,a) = \begin{cases} pop & \text{if } X = a \text{ and } a \in V_T \\ accept & \text{if } X = \# \text{ and } a = \# \\ (a\alpha, i) & \text{if } X \rightarrow a\alpha \text{ is the } i\text{-th rule} \\ error & otherwise \end{cases}$$

For the example grammar, the parsing function is given in Fig. 4.6. Empty entries denote "error" action. The stack contents are usually shown horizontally.



The action of the simple LL(1) parser is shown for the example string shown in Table 4.3.

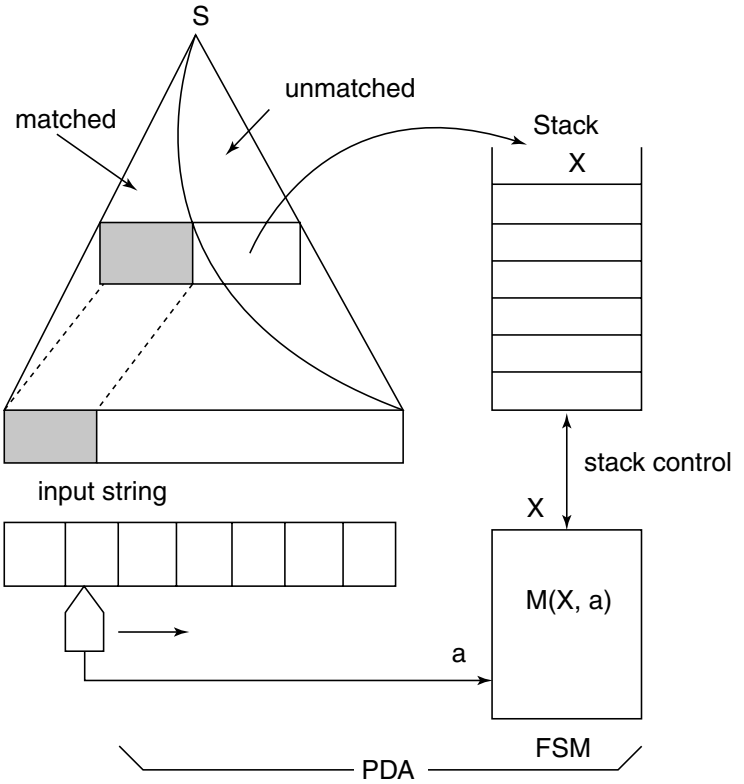


Fig. 4.5 Parsing action with PDA in an LL(1) type parser

a->	a	b	+	*	(	)	#
X							
S	0	0			0		
E	1	3			2		
A			4	5			
a	pop						
b		pop					
+			pop				
*				pop			
(					pop		
)						pop	
#							accept

Fig. 4.6 Parsing function *M*(*X*, *a*) for simple LL(1) grammar

Table 4.3 Simple LL(1) parser: input string $b + b * a\#$

Stack	Input string	Action	Rule applied
S	$\Delta b + b * a\#$	0	$S \rightarrow E\#$
$E\#$	$\Delta b + b * a\#$	3	$E \rightarrow bA$
$bA\#$	$b\Delta + b * a\#$	pop	eliminated
$A\#$	$\Delta + b * a\#$	4	$A \rightarrow +E$
$+E\#$	$+\Delta b * a\#$	pop	eliminated
$E\#$	$\Delta b * a\#$	3	$E \rightarrow bA$
$bA\#$	$b\Delta * a\#$	pop	eliminated
$A\#$	$\Delta * a\#$	5	$A \rightarrow *E$
$E\#$	$*\Delta a\#$	pop	eliminated
$E\#$	$\Delta a\#$	1	$E \rightarrow a$
$a\#$	$a\Delta\#$	pop	eliminated
$\#$	$\#\Delta$	Accept	

Note: Δ denotes the scan pointer.

4.2.4 LL(1) Grammar Without ϵ-rules

We extend the simple LL(1) parser, by removing the restriction that the first symbol on the RHS be in V_t. Surprisingly, such grammars are as much powerful, no more no less, as the Simple LL(1) grammars.

Example

Given a grammar with productions:

```
P = { 0. S'-> S#
      1. S -> ABe
      2. A -> dB
      3. A -> aS
      4. A -> c
      5. B -> AS
      6. B -> b
}
```

Consider input string `adbbeccbee#`. How could we have generated this sentence using the given grammar?

```
S'   => S#            rule 0: only rule
     => ABe#          rule 1: only rule, but no error, as A leads to a,c,d
```

```
=> aSBe#          rule 3 selected: as we want 'a' in front
=> aABeBe#        rule 1: only rule, but no error, as A leads to a,c,d
=> adBBeBe#       rule 2 selected: as we want 'd' in front
=> adbBeBe#       rule 5 rejected: as it would have placed A in front,
                    which would have lead to a,c,d, but we want 'b'
                  rule 6 selected: gives us 'b'
=> adbbeBe#       rule 6 selected: gives us 'b'
=> adbbeASe#      rule 5 selected: rejecting rule 6
=> adbbecSe#      rule 4 selected: we want 'c' in front
=> adbbecABee#    rule 1: only rule, but no error
=> adbbeccBee#    rule 4 selected: we want 'c' in front
=> adbbeccbee#    rule 6 selected: gives us 'b'
```

How can we parse this string? We try as shown in Table 4.4.

Table 4.4 Simple LL(1) parser: input string `adbbeccbce#`

Stack	Remaining input	Rule	Remarks
S#	adbbeccbce#	1	only rule
ABe#	adbbeccbee#	3	as we have 'a' input
aSBe#	adbbeccbee#	pop	a
SBe#	dbbeccbee#	1	only rule
ABeBe#	dbbeccbee#	2	as we have 'd' input
dBBeBe#	dbbeccbee#	pop	d
BBeBe#	bbeccbee#	6	reject 5, as it would have given d, a or c
bBeBe#	bbeccbee#	pop	b
BeBe#	beccbee#	6	reject 5
beBe#	beccbee#	pop	b
eBe#	eccbee#	pop	e
Be#	ccbee#	5	reject 6
ASe#	ccbee#	4	we want 'c'
cSe#	ccbee#	pop	c
Se#	cbee#	1	only rule
ABee#	cbee#	4	we want 'c'
cBee#	cbee#	pop	c
Bee#	bee#	6	we want 'b'
bee#	bee#	pop	b
ee#	ee#	pop	e
e#	e#	pop	e
#	#	accept	

The critical decision was made when we encountered B for the first time; we rejected rule 5 and accepted rule 6, in view of given us $\{d, a, c\}$, but we have a 'b' on the top-of-stack (TOS) to match. Our experimenting with generation of a sentence guided us in parsing the sentence. We had to check what will be the front symbol ultimately get inserted when we replace a NT by another.

Armed with this experience, we now formalize the approach.

A function which tells us what symbol can get inserted in front is called the $FIRST(\beta)$ for some string β. First, we give a working definition for the present, and later we shall have a more thorough definition.

Given some string $\alpha \in G\ (V_T\ \epsilon\ V_N)^*$, the set of Terminal symbols which can be leftmost derived from α:

$$FIRST(\alpha) = \{\omega \mid \alpha \overset{*}{\Rightarrow} \omega \ldots \omega \in V_T\}$$

Using the notation,
T is a Terminal symbol,
N is a non-terminal,
X is a Terminal or non-terminal,
β is a sequence of Terminals or non-terminals,
$t \in FIRST(\beta)$ if t can appear as the first symbol in a derivation starting from β. Recursive definition of $FIRST(\beta)$ is given below:

1. $t \in FIRST(t)$.
2. $FIRST(X) \subseteq FIRST(X\beta)$.
3. $N \overset{*}{\Rightarrow} \epsilon \Rightarrow FIRST(\beta) \subseteq FIRST(N\beta)$.
4. $N \rightarrow Xb \Rightarrow FIRST(X\beta) \subseteq FIRST(N)$.

Note that rule (3) refers to empty string ϵ and is given here in anticipation of the next case of grammar.

For the example grammar: $FIRST(A) = \{d, a, c\}$, $FIRST(B) = \{d, a, c, b\}$, $FIRST(ABe) = \{a, c, d\}$, etc. Note that the following three – $FIRST(dB) = \{d\}$, $FIRST(aS) = \{a\}$ and $FIRST(c) = \{c\}$ – are the disjoint sets for productions from A. Similarly, $FIRST(AS) = \{d, a, c\}$ and $FIRST(b) = \{b\}$ are disjoint sets for productions from B.

Definition 4.2.1 (A LL(1) grammar without ϵ-rules) *A CFG without ϵ-rules is a LL(1) grammar if for all production rules of the form: $A \rightarrow \alpha_1 / \alpha_2 / \ldots / \alpha_n$ the sets $FIRST(\alpha_1)$, $FIRST(\alpha_2)$, …, $FIRST(\alpha_n)$ are pair-wise disjoint, i.e. $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$, for $i \neq j$.*

We need to use a modified parsing function:

$$M : \{V \cup \#\} \times \{V_T \cup \#\} \rightarrow \{(\beta, i),\ pop, accept, error\}$$

$$M(X,a) = \begin{cases} pop & \text{if } X = a \text{ and } a \in V_T \\ accept & \text{if } X = \# \text{ and } a = \# \\ (\beta, i) & \text{if } X \rightarrow \beta \text{ is the } i\text{-th rule and } a \in FIRST(\beta) \\ error & otherwise \end{cases}$$

The parsing table for the example grammar is given in Fig. 4.7.

a->	a	b	c	d	e	#
X						
S	(ABe,1)		(ABe,1)	(ABe,1)		
A	(AS,3)		(c,4)	(dB,2)		
B	(AS,5)	(b,6)	(AS,5)	(AS,5)		
a	pop					
b		pop				
c			pop			
d				pop		
e					pop	
#						accept

Fig. 4.7 Parsing function $M(X, a)$ for LL(1) grammar w/o ϵ-rules

4.2.5 LL(1) Grammars with ϵ-rules

The definition of the $FIRST(\alpha)$ set is to be extended to include zero length strings:

$$FIRST(\alpha) = \{\omega \mid \alpha \overset{*}{\Rightarrow} \omega \ldots \text{ and } |\omega| \leq 1,\ \omega \in (V_T \cup \{\varepsilon\})\},\ \text{so that } |\omega| \text{ is 0 or 1.}$$

Example

Consider the following grammar:

```
0.  S' -> A#
1.  A  -> iB = e
2.  B  -> SB
3.  B  -> <empty>
4.  S  -> [eC]
5.  S  -> .i
6.  C  -> eC
7.  C  -> <empty>
```

Here, we have the following FIRST() functions:

```
FIRST(A#) = {i}          FIRST(<empty>) =    {<empty>}
FIRST(iB = e) = {i}      FIRST(eC) = {e}     FIRST(.i) = {.}
FIRST(SB) = {[.}         FIRST([eC]) = {[}
```

Note that in this grammar we have two *nullable non-terminals* – B and C. We use a name $V_{N\epsilon}$ for the set of Nullable-NT and say that $B, C \in V_{N\epsilon}$.

For the grammar without ϵ-rules, we selected productions based on some $a \in FIRST(\beta)$, where there was a production $B \rightarrow \beta$. Here, in a grammar with ϵ-rules, we have $B \rightarrow A \ldots$ and $A \rightarrow \epsilon$. Then our test should be modified – we should check which symbol follows A in some production $B \rightarrow AG, \ldots A \rightarrow \epsilon$. Then to find $M(B, a)$ we should see if $G \rightarrow a \ldots$. That means we should define a $FOLLOW(A)$ set.

For a Nullable-NT A, we define

$$FOLLOW(A) = \{\omega \mid S' \overset{*}{\Rightarrow} \alpha A \gamma \text{ and } \omega \in V_T \text{ and } \omega \in FIRST(\gamma)\}.$$

Thus for our example grammar, $FOLLOW(C) = \{]\}$ (see rule (4) below) and $FOLLOW(B) = \{=\}$. Suppose the input string is `i[e]=e#`. Its leftmost derivation is

```
A# => iB = e# => iSB = e# => i[eC]B = e# => i[e]B = e# => i[e] = e#
```

This gives us a hint about how to define the FOLLOW() set table.

Finding FIRST(α) Set

A more complete, recursive, formal definition of FIRST() function, useful for developing algorithms is given below:

1. $FIRST(a) = \{a\}$, if $a \in (V_T \cup \{\epsilon\})$.
2. If there is a production $A \to a\alpha$, where $a \in V_T$ and $\alpha \in V^*$, then $a \in FIRST(A)$.
3. If there is a production $A \to B\alpha$, where $B \in V_N$ and $\alpha \in V^*$, then $\forall b \in FIRST(B)$, $b \in FIRST(A)$, i.e. $FIRST(B) \subseteq FIRST(A)$.
4. Generalization of rule (3): If there is a production $A \to \beta X \alpha$, where $\beta \in V^*_{N\varepsilon}$ (i.e. contains Nullable-NT only), $X \in (V_N \cup V_T)$ and $\alpha \in V^*$, then $FIRST(A)$ includes $FIRST(X) \cup FIRST(\beta)$

Note that $\epsilon \in FIRST(B)$ if $B \in V_{N\epsilon}$.

Finding FOLLOW(a) Set

A recursive definition of FOLLOW() set is:

1. If S is the starting symbol, then $\# \in FOLLOW(S)$.
2. If there is a production of the form $A \to \ldots BX \ldots$, where $B \in V_N$ and $X \in V$, then $FIRST(X) \subseteq FOLLOW(B)$.
3. Generalization of rule (2): If there is a production of the form $A \to \ldots B\beta X \ldots$, where $B \in V_N$, $\beta \in V^*_{N\epsilon}$ and $X \in$ V, then $FOLLOW(B)$ includes $FIRST(X) \cup FIRST(\beta)$.
4. If there is a production of the form $A \to \ldots B$, where $B \in V_N$, then $\forall a \in FOLLOW(A)$, $a \in FOLLOW(B)$, i.e. $FOLLOW(A) \subseteq FOLLOW(B)$.
5. Generalization of rule (4): If there is a production of the form $A \to \ldots B\beta$, where $B \in V_N$ and $\beta \in V^*_{N\epsilon}$, then $FOLLOW(A) \subseteq FOLLOW(B)$, $FIRST(\beta) \subseteq FOLLOW(B)$.

There is one exception to be noted: consider a production of the form $A \to \ldots AB$. Here, $FIRST(B) \subseteq FOLLOW(A)$ and rule (4) says $FOLLOW(A) \subseteq FOLLOW(B)$. Then $FIRST(B) \subseteq FOLLOW(B)$. In this case, such $FIRST(B)$ should be subtracted from $FOLLOW(B)$ while testing for LL(1) property.

Table 4.5 shows parsing of the sentence `i[e] = e#`.

Definition 4.2.2 (LL(1) grammar) *A CFG is LL(1) if for each pair of rules for some non-terminal* A, $A \to \alpha$ and $A \to \beta$,

$$FIRST(\alpha\, FOLLOW(A)) \cap FIRST(\beta FOLLOW(A)) = \varnothing$$

or, for all rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

Table 4.5 Parsing of example sentence `i[e] = e#`

Stack	Input string	Action/Remarks
A#	i[e] = e#	rule 1
iB = e#	i[e] = e#	pop i
B = e#	[e] = e#	reject 3, select 2: because '[', $B \in V_{N\epsilon}$
SB = e#	[e] = e#	reject 5, select 4
[eC]B = e#	[e] = e#	pop [
eC]B = e#	e] = e#	pop e
C]B = e#	] = e#	] $\notin FIRST(C)$, select 7
]B = e#	] = e#	pop]
B = e#	=e#	= $\notin FIRST(B)$, select 3
=e#	=e#	pop =
e#	e#	pop e
#	#	Accept

1. $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$ *for* $i \neq j$, *and further, if* $\alpha_i \overset{*}{\Rightarrow} \epsilon$, *then*
2. $FIRST(\alpha_j) \cap FOLLOW(A) = \varnothing, \forall j \neq i$.

The parsing function M is now defined as

$$M : \{V \cup \#\} \times \{V_T \cup \#\} \rightarrow \{(\beta, i),\ pop,\ accept,\ error\}$$

$$M(X,a) = \begin{cases} pop & \text{if } X = a \text{ and } a \in V_T \\ accept & \text{if } X = \# \text{ and } a = \# \\ (\beta,\ i) & \text{if } X \rightarrow \beta \text{ is the } i\text{-th rule and } a \in FIRST(\beta) \\ & \text{or, if } X \rightarrow \beta \text{ is the } i\text{-th rule and } a \in FOLLOW(X) \text{ and} \\ & \epsilon \in FIRST(\beta),\ \text{i.e. } X \in V_{N\epsilon} \\ error & otherwise \end{cases}$$

4.3 Bottom-up Parsing

We shall now discuss another approach to parsing – Bottom-up parsing, which starts with the given sentence to be parsed, detects a *handle* while left-to-right scanning and reduces the sentential forms by replacing the handle by an NT according to the grammar rules. The key issue here will be to detect the handle reliably (Figs. 4.1 and 4.3).

We shall start by discussing a general approach to what is known as Shift/Reduce parsing and Operator Precedence parsing. Then we define the context-free grammars, which are suitable for Bottom-up parsing methods. Such grammars are called LR(k) grammars. The nomenclature LR(k) has the following significance:

$$\overbrace{\textit{Left-to-right scan}}^{L}\ \overbrace{\textit{Rightmost derivation}}^{R}\ \overbrace{\textit{Number of look-ahead symbols}}^{(k)}$$

Parsers built based on LR(k) grammars use the productions in reverse, as they parse by reducing the sentential forms.

4.3.1 Shift/Reduce Parsing

A general approach to Bottom-up parsing is called Shift/Reduce parsing. Referring to the grammar:

```
G = <{I,D,L}, {a-z, 0-9}, I, P>, with productions
P = { 1. I -> L
      2. I -> IL
      3. I -> ID
      4. L -> a | b | ... |z
      5. D -> 0 | 1 | ... |9
    }
```

and Fig. 4.3, we can see that while parsing the given sentence, we reduce portions of the sentence as much as we can and then look for other reducible portions. This gives us a number of partial trees during the incomplete parsing process. For example, while parsing the sentence "x2", we go up to reducing 'x' to L and then to I. We then have to suspend that branch of reduction temporarily and see if we can reduce '2'. After '2' is reduced to D, we are now ready to reduce 'ID' to I. This "suspend temporarily one activity while we take up and complete some other activity" seems like a good candidate for our familiar stack operations. So why not store the roots of all the partial trees on a stack? In fact the stack we want to use is the stack of the PDA model of a Bottom-up parser.

Thus, the parsing can proceed like this:

```
PDA-stack content   Remaining input string   Action done
#                   x  2   #                 start of parsing
# x                    2   #                 Shift (read) x
# L                    2   #                 Reduce x to L by rule 4
# I                    2   #                 Reduce L to I by rule 1
# I 2                      #                 Shift 2
# I D                      #                 Reduce 2 to D by rule 5
# I                        #                 Reduce ID to I by rule 3 Accept
```

There are several parsing algorithms of Shift/Reduce type – the LR(k) parsers and Operator Precedence Parsers.

Comparing the use of the stack in Shift/Reduce parser with use in an LL(1) type of parser, we note that in LL(1) parser, the unmatched part of the sentential form during the Top-down parsing process was stored on the stack. In case of Shift/Reduce parsers, the roots of the trees which get created during the process of reduction are stored on the stack.

4.3.2 Operator Precedence Parsing

Operator precedence parsing is a simple Bottom-up parsing method which can be used if the grammar is expressed in a particular form, known as *operator grammar*. For such a grammar, a Shift/Reduce kind of parser can be built using this technique.

Operator Grammar

First, we define the operator grammar.

Definition 4.3.1 (Operator Grammar) *An operator grammar is the one in which no right-hand side of any production contain two adjacent non-terminals.*

For example, grammar G given above is not an operator grammar, because ID appears in one RHS. On the other hand, the arithmetic expression grammar G_1' is an operator grammar.

For such a grammar we can define *operator precedence relations* between the Terminal symbols of an operator grammar as follows:

- $x \doteq y$ for $x, y \in V_T$ if $\exists$ a production $A \rightarrow uxyv$ or $A \rightarrow uxByv$, where $A, B \in V_N$ and $u, v \in (V_T \cup V_N)^*$
- $x \gtrdot y$ for $x, y \in V_T$ if $\exists$ productions $A \rightarrow uByv$ and $B \overset{+}{\Rightarrow} wx$ or $B \overset{+}{\Rightarrow} wxC$, where $A, B, C \in V_N$ and $u, v, w \in (V_T \cup V_N)^*$
- $x \lessdot y$ for $x, y \in V_T$ if $\exists$ a production $A \rightarrow uxBv$ and $B \overset{+}{\Rightarrow} yw$ or $B \overset{+}{\Rightarrow} Cyw$, where $A, B, C \in V_N$ and $u, v, w \in (V_T \cup V_N)^*$

Consider the definition of $x \gtrdot y$ – a look at the productions for A and B shows that operation x is to be done before the operation y is done. Thus, x has a precedence higher than that of y. Similar comments apply to the other two relations. Note that the precedence is not by the lexical order in the string, but rather on which action will be done first.

For example, referring to the arithmetic expression grammar G_1', ($\doteq$) and $* \gtrdot +$.

We can interpret the relations $x \doteq y$, $x \gtrdot y$ and $x \lessdot y$ as same, higher and lower precedences, but we have to be careful while applying these ideas to a language like C. Remember that in C, in an expression like `c + d + e`, the leftmost '+' has higher precedence compared to the other '+'. Thus, the precedence relation is $+ \gtrdot +$ and not $+ \doteq +$.

Definition 4.3.2 (Operator precedence grammar) *It is an operator grammar for which only one of the relations $\lessdot$, $\doteq$, $\gtrdot$ holds for each pair of Terminal symbols*.

If we want to control parsing via the operator precedence, then we would have to construct a table for relations between all Terminals taken by pairs. That would be a rather large table. Instead, we can generally represent the precedence by two functions `lprec()` and `rprec()`, with properties:

$$a \lessdot b \text{ iff } lprec(a) < rprec(b)$$

$$a \gtrdot b \text{ iff } lprec(a) > rprec(b)$$

$$a \doteq b \text{ iff } lprec(a) = rprec(b)$$

It is possible to implement an RDP even for a left-recursive arithmetic expression grammar using the concept of operator precedence.

In order that a Bottom-up parser can easily be built based on the concept of operator precedence, one further condition needs to be satisfied by the grammar – there should not be any ϵ as an RHS.

There are some disadvantages of operator precedence parsing:

1. It is difficult to handle operator tokens like a minus sign, which can be used both as a unary and as a binary operator, and requires use of two different values of precedence for these two cases.
2. One cannot be sure if the parser accepts exactly the language specified by the grammar.
3. The method is applicable to only a small class of CFG languages.

On a positive note, the method allows one to hand code efficient Shift/Reduce parser, whenever it is applicable.

There are two methods for determining the precedence relations between various Terminal symbols in a grammar:

1. Intuitive method based upon the traditional arithmetic concepts of precedence and associativity of operators. For example, we generally take the multiplication operation, denoted by '*', to have a higher precedence compared to that of addition operation, denoted by '+'. Thus, we can make $* \gtrdot +$ and $+ \lessdot *$.
2. First, construct an unambiguous grammar for the language, which reflects correct precedence and associativity in a parse tree of the language. Then use an algorithm to convert this grammar into an operator precedence relations table. There is a possibility with this method though that the parser may not exactly recognize the language intended.

We shall now give an operator precedence parsing algorithm, developed using the first method. Consider an example grammar:

```
E -> E + T | T
T -> T * F | F
F -> a
```

Suppose we use $ to mark each end of the input string being parsed. The assumed precedence relations are given in Table 4.6.

Table 4.6 Precedence relations for the example grammar

	a	+	*	$
a		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

The idea is to capture a handle between $\lessdot$ and $\gtrdot$, by inserting the relations symbols between each Terminals in a sentential form. For example, sentential form `a + a * a`, after insertion of the relations, looks like: $\lessdot a \gtrdot + \lessdot a \gtrdot * \lessdot a \gtrdot$ $.

Note that the first handle 'a' is captured between the first $\lessdot$ from the left and first $\gtrdot$. As the actual Non-Terminals do not control the parsing, only one of them is loaded on the stack to represent all of them. The parsing algorithm is given in Algorithm 4.3.1.

Algorithm 4.3.1: Operator precedence parsing

```
/* Initially the stack contains $, string w$ input*/
ip ← pointer to first symbol of w$;
while True do
  if TOS = $ and (ip) = $ then
    return;
  else
    a topmost Terminal and (ip) = b;
    if a ⋖ b or a ≐ b then
      push(b);
      ip++;
    else
      if a ⋗ b then
        /* Reduce */
        repeat
          pop;
        until TOS ⋖ most recently popped;
      else
        ERROR;
      end
    end
  end
end
```

It should be noted here that Algorithm 4.3.1 is very similar to the one used in the first compiler for one of the earliest High Level languages – FORTRAN.

Unary operators are handled by keeping their precedence the highest, i.e. for any operator θ, $\theta \lessdot \neg$, where $\neg$ is a representative unary operator. If same atomic symbol is used for unary and binary operations, e.g. minus '$-$' sign, then it is best to let the Scanner detect and replace the unary minus with a unique token, separate from binary minus.

4.3.3 LR Grammars

The following are the general properties of LR grammars:

- Almost all unambiguous CFG are LR grammars.
- The set of LR grammars contain as a subset the set of LL grammars.
- They scan from left-to-right and construct rightmost derivations in the reverse direction.
- They make parsing decisions based on contents of the parse-stack and next k look-ahead symbols. Note that the parser model being PDA, a stack is required to be there.
- They have *valid-prefix* property, which detects errors, if any, at earliest possible step during parsing. The valid-prefix property ensures that if the parser cannot detect any error in prefix X in a program XY, then there is a valid program XW for some W.

- LR(k) parsers are in general more complicated compared with LL(k) parsers. LL(k) for $k > 1$ is very tedious to construct, so the approach generally taken is try LL(0) and LL(1), if not possible, then try in the order of increasing complexity – LR(0), SLR(1), LALR(1) and LR(1).
- Compiler writing tools `yacc` and `bison` are designed based on LALR(1) techniques; `ANTLR` is based on LL(*) approach.

Concepts and Terminology

Given a grammar G, having starting symbol S. Suppose we want to parse a given string x

$$S \underset{R}{\Rightarrow} \alpha_1 \underset{R}{\Rightarrow} \alpha_2 \underset{R}{\Rightarrow} \ldots \underset{R}{\Rightarrow} \alpha_m = x$$

where $\alpha_1 \underset{R}{\Rightarrow} \alpha_2$ is of the form $\phi B t \underset{R}{\Rightarrow} \phi\beta t$, $t \in V_T^*$ and $B \rightarrow \beta$ is a production in the grammar. The string β will be used as a handle. LA is the look-ahead symbol within t.

ϕ	B	t
ϕ	β	$\boxed{LA}\ t$

Then the grammar G is LR(k) if for any given x, at each step of derivation, the handle β can be detected by examining $\phi\beta$ and scanning at most k symbols of the remaining string t.

Viable prefix Viable prefix of a sentential form $\phi\beta t$, where β denotes the handle, is any prefix or head-string of $\phi\beta$, i.e. if $\phi\beta = u_1\, u_2 \ldots\ u_r$, then $u_1\, u_2 \ldots u_i$, $1 \leq i \leq r$ is a viable prefix. It cannot contain symbols to the right of the handle, i.e. from string t.

Example

Consider a grammar G_2 having productions:

```
S -> E #
E -> T | E + T | E * T
T -> a | ( E )
```

Consider a sentential form `E * (a + a)#` in the rightmost derivation step:

$$E * (T + a)\# \Rightarrow E * (a + a)\#$$

Here,

$$\underbrace{E*(}_{\phi}\ \underbrace{T}_{B}\ \underbrace{+a)\#}_{t} \Rightarrow \underbrace{E*(}_{\phi}\ \underbrace{a}_{\beta}\ \underbrace{+a)\#}_{t}$$

and the viable prefixes are E, $E*$, $E * ($ and $E * (a$.

Prof. Donald Knuth, in his paper [Knu65], in 1965, showed that the set of all the viable prefixes of all the rightmost derivations for a particular grammar is a regular language and as such can be recognized by an FSM. This FSM becomes the controller in the PDA model of the LR(k) parser. Any practical and useful grammar will usually generate an infinite language and will have an infinite set

of viable prefixes. Thus, a key step in constructing the parser is to develop the FSM from just the grammar specifications. The FSM is constructed by essentially partitioning the viable prefix set into equivalent classes.

The FSM transition diagram is given in Fig. 4.8. Due to the nature of the FSM transitions, we shall call this an LR(0) parser and the grammar an LR(0) grammar.

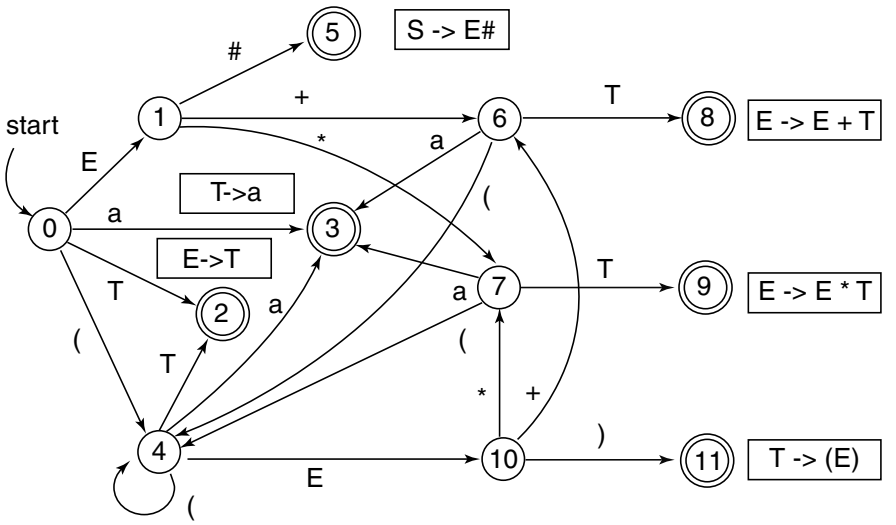


Fig. 4.8 FSM for a parser for the example grammar

4.3.4 FSM for an LR(0) Parser

We shall take the FSM for the example grammar and use it to discuss features of the FSM for an LR(0) parser.

The starting state is state 0. There are two types of states:

Read or Shift: A Terminal or Non-Terminal symbol and take a state transition induced by that symbol.

Reduce: Recognize a handle so that it can be replaced by an NT.

We shall subsequently see the detailed operations performed during these two states. As in Fig. 4.8, read/shift states are 0, 1, 4, 6, 7, 10, and reduce states are 2, 3, 5, 8, 9, 11.

The viable prefixes associated with state 3 are a, $E^*\ a$, $E + a$, $(a$, $((E^*\ a$, etc. The sets of such viable prefixes, in general, are infinite. A closer inspection will show that state 3 is associated with all viable prefixes ending in 'a'. Similar statements can be made about other states. It is suggested that the reader writes down a few viable prefixes for each of the final states of the FSM.

The FSM reaches a final state when a particular production in the grammar is successfully detected. As shown in Fig. 4.8, the productions are written near each of the final states.

The stack of the PDA has a pair as each entry:

$X \in (V_T \cup V_N)$	state entered on detecting X

The top-of-stack shows the current state of the FSM. The operation of the PDA is as follows:

1. Initially, the stack contains $\boxed{S \mid 0}$, S is the start symbol in the grammar and 0 is the starting state of FSM.
2. If in a *read/shift* state: stack the current symbol, perform the transition and stack the new state at which arrived (see Fig. 4.9).
3. If in *reduce* state associated with a production $B \rightarrow \beta$, pop the top $|\beta|$ entries, i.e. the number of entries popped up is equal to the length of string β. If $B = S$, then accept and stop, otherwise the *uncovered* state and the NT symbol on the LHS of the production rule, i.e. B, determine the next action, then stack $\boxed{B \mid \text{next state}}$ (see Fig. 4.10).

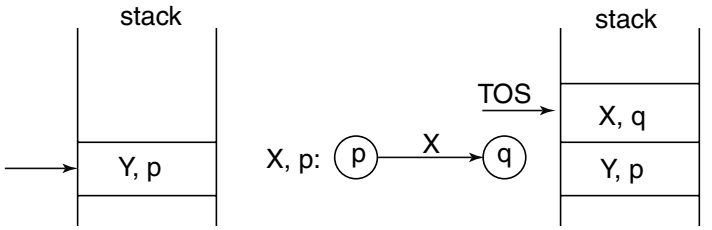


Fig. 4.9 PDA: read/shift operation

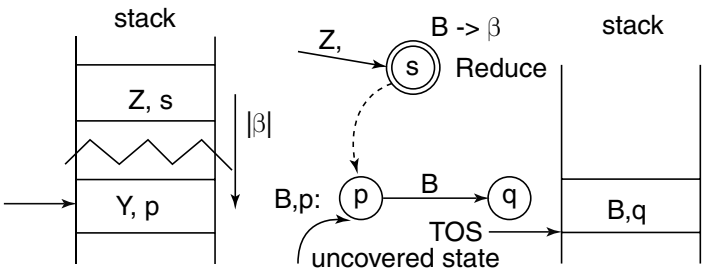
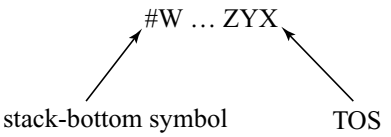


Fig. 4.10 PDA: reduce operation

The LR(0) parser actions for the example string `a * (a + a)#` are given in Table 4.7. For a Bottom-up parser, the stack bottom is on left, stack grows to the right and TOS is the rightmost symbol.



4.3.5 Design of the FSM for an LR(0) Parser

How to design the FSM for an LR(0) parser? As the nomenclature indicates, there is no look-ahead. Each state of the FSM is associated with a set of *viable prefixes*, which will, in general, be an infinite set. These sets are obtained by partitioning the total set of viable prefixes in terms of equivalence classes.

Table 4.7 LR(0) parser actions for the string `a * (a + a)#`

Stack	Input	Viable prefix	Handle
$S0$	$a\ *(a+a)\#$		
$S0\ a3$	$*(a+a)\#$	a	a
$S0\ T2$	$*(a+a)\#$	T	T
$S0\ E1$	$*(a+a)\#$	E	
$S0\ E1\ *7$	$(a+a)\#$	$E*$	
$S0\ E1\ *7\ (4$	$a+a)\#$	$E*($	
$S0\ E1\ *7\ (4\ a3$	$+a)\#$	$E\ *\ (a$	a
$S0\ E1\ *7\ (4\ T2$	$+a)\#$	$E\ *\ (T$	T
$S0\ E1\ *7\ (4\ E10$	$+a)\#$	$E\ *\ (E$	
$S0\ E1\ *7\ (4\ E10+6$	$a)\#$	$E\ *\ (E+$	
$S0\ E1\ *7\ (4\ E10+6\ a3$	$)\#$	$E\ *\ (E+a$	a
$S0\ E1\ *7\ (4\ E10+6\ T8$	$)\#$	$E\ *\ (E+T$	$E+T$
$S0\ E1\ *7\ (4\ E10$	$)\#$	$E\ *\ (E$	
$S0\ E1\ *7\ (4\ E10)11$	$\#$	$E\ *\ (E)$	(E)
$S0\ E1\ *7\ T9$	$\#$	$E\ *\ T$	$E\ *\ T$
$S0\ E1$	$\#$	E	
$S0\ E1\ \#5$		$E\#$	$E\#$
$S0$	Accept	Accept	

Definition 4.3.3 (Item) *An item for a grammar G is a marked production rule of the form $[A \rightarrow \alpha_1 \bullet \alpha_2]$, where $A \rightarrow \alpha_1\ \alpha_2$ is the original production and $\bullet$ is the marker.*

Note that the marker indicates the position reached in the sentential form while scanning.

For example, with the production `E -> E + T`, we have the following four items possible: $[E \rightarrow \bullet E + T]$, $[E \rightarrow E\bullet + T]$, $[E \rightarrow E + \bullet T]$ and $[E \rightarrow E + T\bullet]$.

Definition 4.3.4 (Valid item) *An item is valid for a viable prefix $\phi\alpha_1$ if and only if*

$$\exists S \overset{*}{\underset{R}{\Rightarrow}} \phi A t \Rightarrow \phi\alpha_1\alpha_2 t \quad \text{for } [A \rightarrow \alpha_1 \bullet \alpha_2]$$

There may be many valid items for a viable prefix. For example, the items $[E \rightarrow E* \bullet T]$, $[T \rightarrow \bullet\ a]$, $[T \rightarrow \bullet\ (E)]$ are all valid for the viable prefix `E*` because $S \Rightarrow E\# \Rightarrow E\ *\ T\#$ shows that the item $[E \rightarrow E* \bullet T]$ is valid. Here, we have $\phi = \epsilon$, $\alpha_1 = E*$, $\alpha_2 = T$ and $t = \epsilon$. Similarly, $S \Rightarrow E\# \Rightarrow E\ *\ T\# \Rightarrow E\ *\ a\#$ shows that $[T \rightarrow \bullet a]$ is also a valid item. Here, we have $\phi = E*$, $\alpha_1 = \epsilon$, $\alpha_2 = a$ and $t = \epsilon$.

Further, because

$$S \Rightarrow E\# \Rightarrow E * T\# \Rightarrow E * (E)\#$$

$[T \rightarrow \bullet (E)]$ is also a valid item.

We can thus obtain a finite set of *valid items* for each state. We can now construct the FSM for the example grammar. The construction uses three operations:

1. forming the basis,
2. closure,
3. successor.

Items like $[S \rightarrow \bullet\ \alpha]$ will be associated with the starting state. It denotes the fact that we have not yet encountered any viable prefixes. Thus, the item $[S \rightarrow \bullet\ E\#]$ is associated with the state 0.

This item is called the *basis* item for that state. We should now apply a *closure* or *completion* operation for the state. The rule is:

If we have an item $[A \rightarrow \alpha_1 \bullet X\alpha_2]$, where $X \in V_N$, we include all the items of the form $[X \rightarrow \bullet\ \alpha_3]$ in our item set.

Thus for state 0, the closure operation gives items:

$$[S \rightarrow \bullet E\#] \mapsto [E \rightarrow \bullet\ T]\ [E \rightarrow \bullet\ E + T]\ [E \rightarrow \bullet\ E * T]$$

$$[E \rightarrow \bullet\ T] \mapsto [T \rightarrow \bullet\ a]\ [T \rightarrow \bullet\ (\ E\ )]$$

No further items can be included for this set, so the closure operation ends.

This set will be called set C_0 =

C_0
[S → • E#]
[E → • T]
[E → • E + T]
[E → • E * T
[T → • a]
[T → • (E)]

We now use the *successor* operation to obtain a new state of the FSM. Suppose an item $[A \rightarrow \alpha \bullet X\beta]$ is associated with some state P, where $X \in (V_t \cup V_N)$. Then item $[A \rightarrow \alpha X \bullet \beta]$ is associated with a state Q.

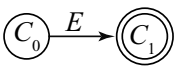
Note that the FSM takes a transition when reading a symbol or while it is in a reduce state. Thus in our example grammar, a successor operation, reduce on E, gives us a basis for a new state 1:

$$[S \rightarrow E \bullet\#]\ [E \rightarrow E \bullet + T]\ [E \rightarrow E \bullet * T]$$

No closure is required as a Terminal symbol follows the marker.

This is set C_1 =

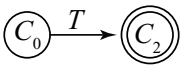
C_1
[S → E • #]
[E → E • + T]
[E → E • * T]



Similarly, $[E \rightarrow \bullet\ T]$ gives us $[E \rightarrow T \bullet]$ and no closure items are generated.

Thus set C_2 =

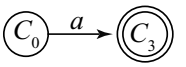
C_2
[E → T •]



Similarly, [$T \rightarrow \bullet\, a$] gives us [$T \rightarrow a \bullet$] and no closure items are generated.

Thus set C_3 =

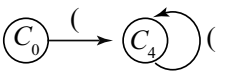
C_3
[T → a •]



[$T \rightarrow \bullet\, (E)$] gives us [$T \rightarrow (\bullet\, E)$] and generates the following closure items: [$E \rightarrow \bullet\, T$] [$E \rightarrow \bullet\, E + T$] [$E \rightarrow \bullet\, E * T$] [$T \rightarrow \bullet\, a$] [$T \rightarrow \bullet\, (E)$]

Thus set C_4 =

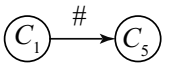
C_4
[E → • T]
[E → • E + T]
[E → • E * T]
[T → • a]
[T → • (E)]



From state C_1 to set C_5 on shifting symbol #, which gives us

set C_5 =

C_5
[S → E # •]

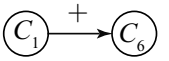


Note that it is possible, if we continue the above process of shift, base and closure, to come up with a state with exactly the same item set as some previous state. We simply ignore this duplicate state and use instead the old set.

From state C_1, on reading/shifting '+' we come to state C_6, i.e. [$E \rightarrow E \bullet + T$] gives [$E \rightarrow E + \bullet\, T$] and the closure is [$T \rightarrow \bullet\, a$] [$T \rightarrow \bullet\, (E)$].

Thus set C_6 =

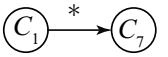
C_6
[E → E + • T]
[T → • a]
[T → • (E)]



Similarly, from state C_1, on reading/shifting '*' we come to state C_7, i.e. [$E \rightarrow E \bullet * T$] gives [$E \rightarrow E * \bullet\, T$] and the closure is [$T \rightarrow \bullet\, a$] [$T \rightarrow \bullet\, (E)$].

Thus set C_7 =

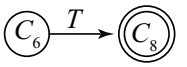
C_7
[E → E * • T]
[T → • a]
[T → • (E)]



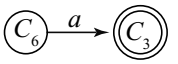
From state C_6, on reading/shifting 'T' we come to state C_8, i.e. [$E \rightarrow E + \bullet\, T$] gives [$E \rightarrow E + T \bullet$] and there is no closure.

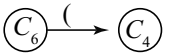
Thus set C_8 =

C_8
[E → E + T •]



Also, the following transitions are done to states already defined:

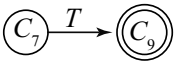




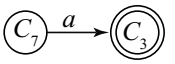
From state C_7, on reading/shifting 'T' we come to state C_9, i.e. [$E \rightarrow E * \bullet T$] gives [$E \rightarrow E * T \bullet$] and there is no closure.

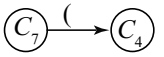
Thus set C_9 =

C_9
[E → E * T •]



Also, the following transitions are done to states already defined:

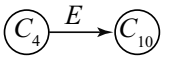




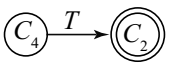
From state C_4, on reading/shifting 'E' we come to state C_{10}, i.e. [$T \rightarrow (\bullet E)$] gives [$T \rightarrow (E \bullet)$] [$E \rightarrow E \bullet + T$] [$E \rightarrow E \bullet * T$] and there is no closure.

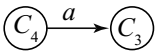
Thus set C_{10} =

C_{10}
[T → (E •)]
[E → E • + T]
[E → E • * T]



Also, the following transitions are done to states already defined:

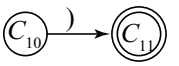




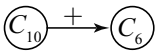
From state C_{10}, on reading/shifting ')' we come to state C_{11}, i.e. [$T \rightarrow (E \bullet)$] gives [$T \rightarrow (E) \bullet$] and there is no closure.

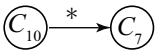
Thus set C_{11} =

C_{11}
[T →(E) •]



Also, the following transitions are done to states already defined:





This completes the generation of the FSM of the PDF from the grammar. If you compare the states and transitions with Fig. 4.8, you will find one-to-one correspondence.

4.3.6 Table-driven LR(0) Parser

We have to now construct the parsing table for the given grammar. We now define two functions:

F() is an action function or mapping; F(state on TOS, current input) $\rightarrow \{S, rk, A, error\}$, where $S =$ shift in a symbol, $rk =$ reduce by production rule k, $A =$ accept the string and error will be denoted by a blank in the table.

G() is the next state or GoTo function; G(state on TOS, symbol in V) $\rightarrow \{q, error\}$, where $q =$ a new state and error will be denoted by a blank in the table.

Even though an LR(0) grammar does not require a look-ahead, we shall use a look-ahead of 1 symbol and construct the tables for $F()$ and $G()$.

Thus, the design of a table-driven LR(0) parser proceeds as follows:

Given an LR(0) finite-state machine represented by a set of item sets, $C = \{C_0, C_1, \ldots, C_m\}$, where $\{0, 1, \ldots, m\}$ are the states of the FSM and 0 is the starting state, repeat the following for each state i:

1. **Compute the action entries *F*()**:
   1. If $[A \rightarrow \alpha \bullet x\beta] \in C_i$, where $x \in (V_T \cup \{\epsilon\})$ and there is a transition from C_i to C_j on x, then $F(i, x) \leftarrow$ shift.
   2. If $[A \rightarrow \gamma \bullet\ ] \in C_i$, where $A \rightarrow \gamma$ is the jth production rule, then $F(i, x) \leftarrow$ reduce j $\forall x \in FOLLOW(A)$.
   3. If $[S \rightarrow \alpha \bullet\ ] \in C_i$, then $F(i, x) \leftarrow$ Accept.
   4. All undefined entries in F-table are blanks, i.e. errors.
2. **Compute the GoTo entries *G*()**:
   1. If there is a transition from C_i to C_j on x, then $G(i, x) \leftarrow j$.
   2. All undefined entries in G-table are blanks, i.e. errors.

The starting state is 0.

The algorithm for a table-driven LR(0) parser (and also for a table-driven SLR parser discussed in Section 4.3.9) is given as Algorithm 4.3.2.

Algorithm 4.3.2: Table-driven LR(0) and SLR parser

```
push 0;
current ← nextchar;
while not EOF do
    if F(TOS, current) = S then
        Shift: temp ← G(TOS, current);
        push(temp);
        current ← nextchar;
        if EOF then goto ERR;
```

```
    else
      if F(TOS, current) = rk then
        reduce: pop | RHS(k) | entries;
        temp ← G(TOS, LHS(k));
        push(temp);
      else
        if F(TOS, current) = A then
          Accept;
          exit;
        else
          error exit;
        end
      end
    end
  end
  ERR: error exit;
```

The parsing table with functions $F()$ and $G()$ is given in Table 4.8.

Table 4.8 Functions $F()$ and $G()$ for example grammar

	$F()$							$G()$								
State	a	$*$	(	)	$+$	#	ϵ	S	E	T	a	$*$	(	)	$+$	#
0	S		S						1	2	3		4			
1		S			S	S						7			6	5
2		$r2$		$r2$	$r2$	$r2$										
3		$r5$		$r5$	$r5$	$r5$										
4	S		S						10	2	3		4			
5							A									
6	S		S							8	3		4			
7	S		S							9	3		4			
8		$r3$		$r3$	$r3$	$r3$										
9		$r4$		$r4$	$r4$	$r4$										
10		S		S	S							7			11	6
11		$r6$		$r6$	$r6$	$r6$										

4.3.7 Parsing Decision Conflicts

Recall that the actions of a Shift/Reduce parser are one of *shift*, *reduce*, *accept* or *error*. A grammar is said to be an LR(0) grammar if rules for *shift* and *reduce* are unambiguous. That is, if a state contains a completed item [$A \rightarrow a \bullet$], then it can contain no other items. If, on the other hand, it also contains a "shift" item, then it is not clear if we should do the reduce or the shift and we have a *shift–reduce conflict*. Similarly, if a state also contains another completed item, say [$B \rightarrow b \bullet$], then it is not clear which reduction to do and we have a *reduce–reduce conflict*.

In general, such conflicts do occur for an LR(1) grammar and we should have methods to *resolve* them.

For shift–reduce conflicts there is a simple solution used in practice – always prefer the shift operation over the reduce operation. This automatically handles, for example, the "dangling else" ambiguity in if-statements. Reduce–reduce problems are not so easily handled.

Dangling Else Problem

Consider a C program:

```
main(){
     int a = 2;
     int b = 2;
     if(a == 1)
       if(b == 2)
         printf("a was 1 and b was 2\n");
     else
      printf("a wasn't 1\n");
}
```

This program did not print anything. When the control constructs are properly indented, we can see what has happened.

```
main(){
     int a = 2;
     int b = 2;
     if(a == 1)
       if(b == 2)
         printf("a was 1 and b was 2\n");
       else
         printf("a wasn't 1\n");
}
```

This is called the "dangling else" problem. With the program in its original layout, the programmer thought the else statement

```
else
      printf("a wasn't 1\n");
```

would be associated with the first if, but it was not. In C (and most other languages), an `else` always associates with the immediately preceding `if` as the second layout of the program shows.

In order to achieve the result that the programmer intended, the program needs to be rearranged as:

```
main(){
     int a = 2;
     int b = 2;
     if(a == 1){
       if(b == 2) printf("a was 1 and b was 2\n");
     } else {
       printf("a wasn't 1\n");
     }
}
```

The "dangling else" problem occurs because of a Shift/Reduce conflict in the grammar rules for the IF-THEN and IF-THEN-ELSE statements:

```
ifstmt : IF '(' cond ')' stmt            {/* generate IF-THEN code */}
       | IF '(' cond ')' stmt ELSE stmt  {/* generate IF-THEN-ELSE code */}
       ;
```

The conflicts can be resolved by one of the following methods:

1. Force one of *shift* or *reduce* in all instances and give corresponding semantic interpretation of the constructs, as it is done in case of IF-THEN-ELSE.
2. Specify and use operator precedences and/or associativity, as it is done in case of arithmetic and logical expressions.
3. Rewrite the grammar to avoid the conflicts.

See Section 4.4.6 for ambiguity and conflict resolution in `yacc`.

4.3.8 Canonical LR(1) Parser

A canonical LR(1) parser or LR(1) parser is an LR parser whose parsing tables are constructed in a way similar to LR(0) parsers except that the items in the item sets also contain a *look-ahead*, 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 -> B C` may look like:

$$[A \rightarrow B\ C \bullet, a]$$

where the "dot" denotes that the parser has read a string corresponding to B and C, with the look-ahead symbol 'a'. Various points regarding LR(1) items to be noted are as follows:

- An LR(k) item is a pair [P, s], where P is a production $A \rightarrow \beta$ with a "dot" at some position in its RHS, and 's' is a look-ahead string of length $\leq k$. Usually, we are concerned with only LR(1) grammars and as such, 's' is just a single symbol.
- $[A \rightarrow \bullet\ B\ C, a]$ means that the input seen so far is consistent with the use of the production $A \rightarrow BC$ immediately after the symbol on the top of the stack.
- $[A \rightarrow B \bullet C, a]$ means that the input seen so far is consistent with the use of the production $A \rightarrow BC$ at this point in the parse and that the parser has already recognized B, i.e. B is on the top of the stack.
- $[A \rightarrow B\ C \bullet, a]$ means that the parser has seen BC and that a look-ahead symbol of 'a' is consistent with reducing to A.

- The look-ahead symbol is carried along to help choose the correct reduction.
- Look-aheads are for bookkeeping, unless item has a "dot" at the right-hand end.
  - It has no direct use in $[A \rightarrow B \bullet C, a]$
  - In $[A \rightarrow B\ C \bullet, a]$, a look-ahead of '$a$' implies a reduction by $A \rightarrow BC$.
  - For a state having items $[A \rightarrow B\ C \bullet, a]$ and $[C \rightarrow D \bullet E, b]$, if look-ahead seen is 'a' then *reduce* to A, if the look-ahead is in $FIRST(E)$ then *shift*.
  - Thus comparatively limited right context is enough to pick up correct action.
  - If we have two LR(1) items of the form $[A \rightarrow C \bullet, a]$ and $[B \rightarrow C \bullet, b]$, we can take advantage of the look-ahead to decide which reduction to use. The same situation would produce a Reduce/Reduce conflict in the SLR(1) approach.
- The notion of validity is slightly different from LR(0). An item $[A \rightarrow B \bullet C, a]$ is valid for a viable prefix "$D\ B$" if there is a rightmost derivation that yields "$D\ A\ a\ w$" which in one step yields "$D\ B\ C\ a\ w$".

Building LR(1) Machine

The LR(1) machine is built following the steps:

Initial item: Begin with the initial item of $[S' \rightarrow \bullet S, \$]$, where \$ is a special symbol denoting the end of the string.

Closure: If $[A \rightarrow A' \bullet B\ B', a]$ belongs to the set of items and $B \rightarrow C$ is a production of the grammar, then add an item $[B \rightarrow \bullet C, b]$ for all b in $FIRST(B'a)$.

Goto: A state containing an item $[A \rightarrow A' \bullet X\ B', a]$ will move to a state containing $[A \rightarrow A'X \bullet B', a]$ with label X.

Shift actions: If $[A \rightarrow A' \bullet b\ B', a]$ is in state I_k and I_k moves to I_m with label 'b', then add the action $[k, b]$ = "shift m".

Reduce actions: If item $[A \rightarrow A' \bullet, a]$ is in state I_k, then add action "Reduce by $A \rightarrow A'$ " to action $[k, a]$. Note that we do not use information from $FOLLOW(A)$ anymore.

LR(1) parsers can deal with a very large class of grammars but their parsing tables are often very big.

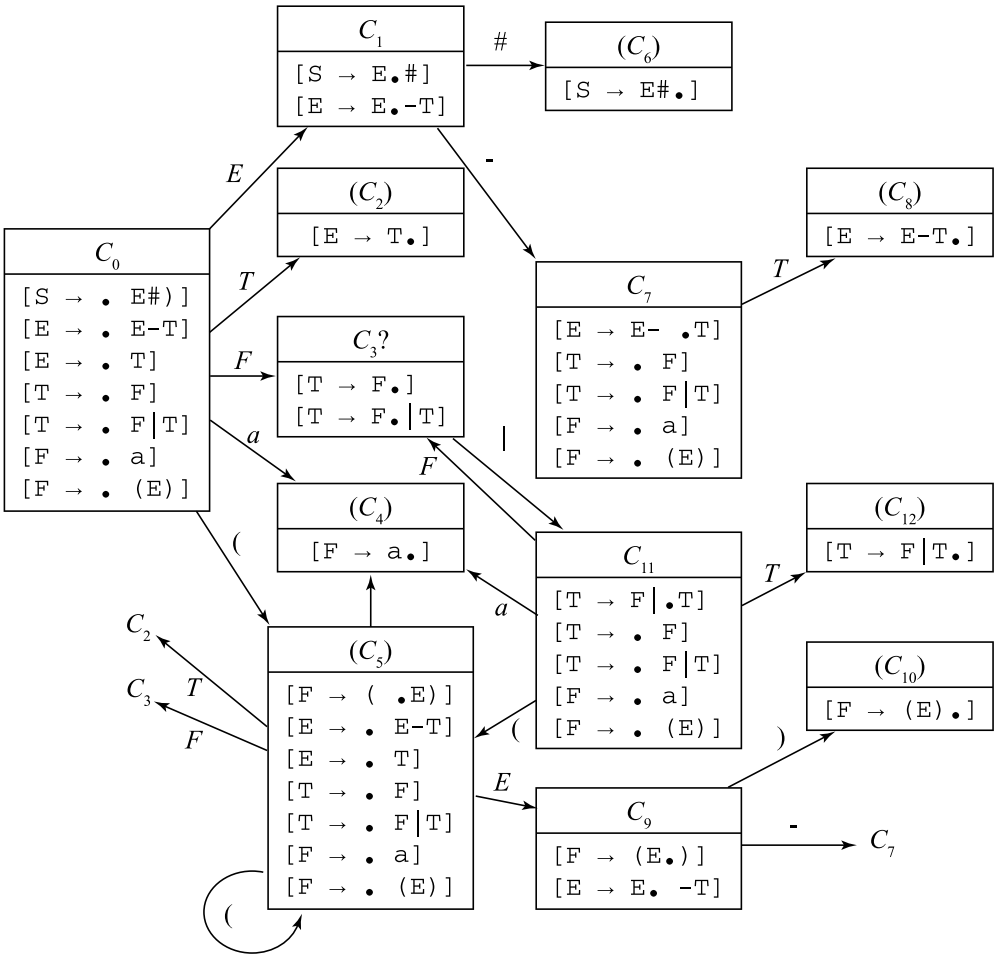
4.3.9 SLR(1) Parser

A simplified version or simple LR(1) parser is called SLR(1) parser. It can be designed from the LR(0) item sets and some additional, easily available information, but all LR(1) grammars may not have an SLR(1) parser possible.

Consider an example grammar:

```
0. S -> E #          4. T -> F

1. E -> E - T        5. F -> ( E )

2. E -> T            6. F -> a

3. T -> F | T
```

The item sets for this grammar are:



The state C_3 is an *inadequate* state, as it has one *completed* item and at least one more item – completed, which will give *Reduce/Reduce* conflict, or – incomplete, which will give *Shift/Reduce* conflict.

Such an inadequate state indicates a local ambiguity in the grammar.

Definition 4.3.5 (LR(0) Grammar) *If the FSM obtained from the LR(0) item sets does not have any inadequate state, it is an LR(0) grammar.*

Such local ambiguity can possibly be resolved by some simple means. For example, for state 3:

C_3?
[T → F •]
[T → F • |T]

If we elect to reduce at C_3, then F will be reduced to T. In the sentential form at this stage, several symbols can possibly follow T. They are $FOLLOW(T) = \{\#, -,)\}$. Since this set is disjoint with respect to the set $\{|\}$, we can resolve the conflict at state 3 by one symbol look-ahead. If the look-ahead symbol is any of $\{\#, -,)\}$, we reduce, if it is $\{|\}$ we shift.

When in a case of a grammar such simple look-ahead technique works, we have an SLR(1) grammar. The algorithm for SLR(1) construction is the same as that for LR(0) grammar, as given previously. If step 1, i.e. for a particular 'x' $F(i, x)$, is both shift and reduce or two or more reduces, then it is not an SLR(1) grammar.

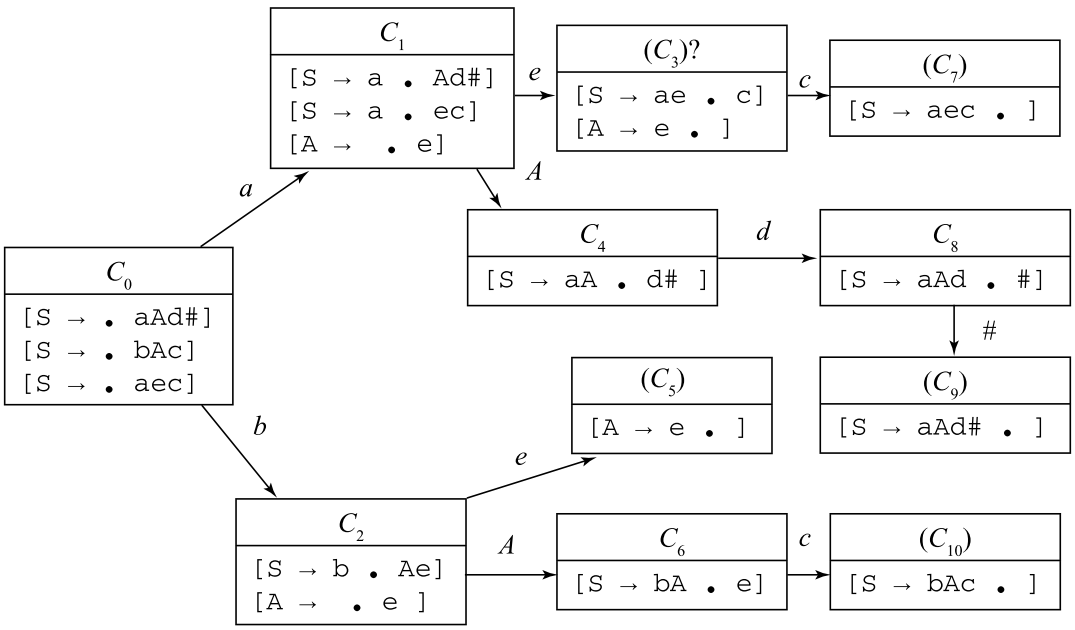
The table-driven parser algorithm given as Algorithm 4.3.2 also applies to a SLR(1) parser.

Examples

We shall now consider a grammar candidate for SLR(1) parsing strategy.

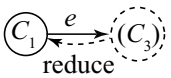
```
1. S -> a A d #
2. S -> b A c
3. S -> a e c
4. A -> e
```

The item sets can be easily computed as shown below.

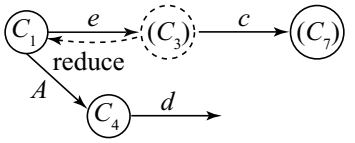


One can see that state 3 is inadequate, as it has both a completed item and an incomplete item. If we decide to reduce at this stage, we shall replace 'e' with A. Then we have to check $FOLLOW(A) = \{c, d\} \cap \{c\} \neq \varnothing$. Therefore, SLR(1) method of resolving this conflict cannot be applied here and this grammar is not an SLR(1) grammar.

If we were to reduce at state 3, we shall unstack one symbol from the stack and *uncover* state 1.



Now we have to consider what follows A in the sentential form if we take transition from state 1 to state 4. This qualified $FOLLOW()$ we write as $FOLLOW(1, A) = \{d\} = LA(3, A \rightarrow e)$. As $FOLLOW(1, A) = \{d\} \cap \{c\} = \varnothing$, so if we take a look-ahead of 1 symbol and see 'd' after 'e', we can reduce 'e' to A; otherwise, we should shift on 'c'.



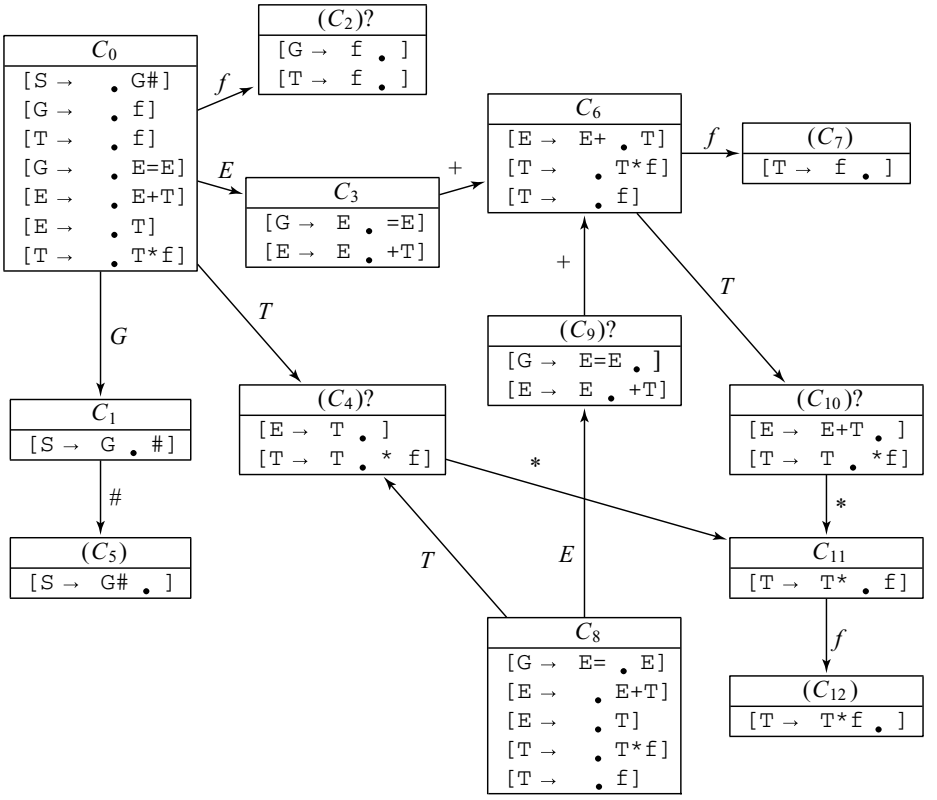
The technique we have just illustrated is known as look-ahead LR(1) or LALR(1) technique, which is discussed more fully in the next sections.

4.3.10 Conflict Situations

Consider another example grammar:

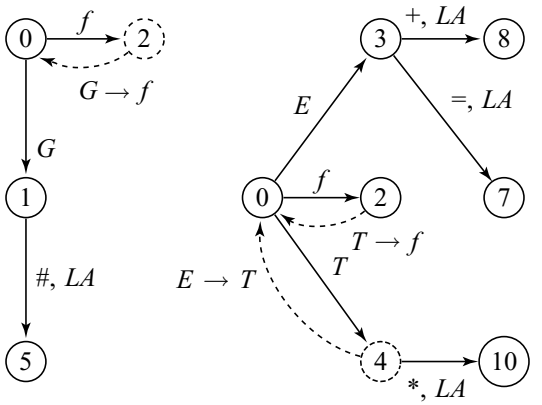
```
0. S -> G #          4. E -> E + T
1. G -> E = E        5. T -> f
2. G -> f            6. T -> T * f
3. E -> T
```

The LR(0) item sets are the following:



State 4 is inadequate, as there is Shift/Reduce conflict, but it can be resolved under SLR(1) technique, as $FOLLOW(E) = \{+, =, \#\}$ which is disjoint with $\{*\}$. Similarly, inadequate states 9 and 10 can also be resolved under SLR(1) technique, as $FOLLOW(G) = \{\#\} \cap \{+\} = \varnothing$ and $FOLLOW(E) = \{+, =, \#\} \cap \{*\} = \varnothing$, respectively.

On the other hand, if we consider the inadequate state 2, it has Reduce/Reduce conflict. The FOLLOW sets of G and T are $FOLLOW(G) = \{\#\}$ and $FOLLOW(T) = \{\#, =, +, *\}$, and since these two sets are not disjoint, we cannot resolve the conflict under SLR(1) techniques. As it is known that this grammar is LR(1), we should be able to parse it with a look-ahead of 1 symbol. For this we shall have to do more detailed analysis of the conflict and look-aheads.



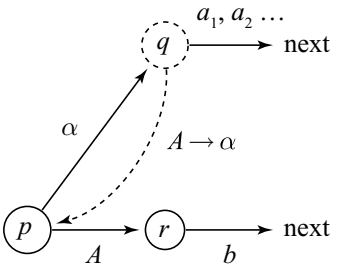
In the partial state diagrams shown above, the look-ahead set if we decide to reduce by $G \rightarrow f$ at state 2 is $LA(2, G \rightarrow f) = \{\#\}$.

On the other hand, what is the situation if we decide to reduce by $T \rightarrow f$ at state 2? State 4 itself is a Shift/Reduce conflict state and so one of the look-aheads there is $\{*\}$. If we take reduce operation at state 4, then we arrive at state 3 and the look-aheads are $\{+, =\}$. Thus, $LA(2, T \rightarrow f) = \{*, +, =\}$. We see that the Reduce/Reduce state 2 is resolvable after all – if the look-ahead symbol is '#', we reduce to G and if it is one of $\{*, +, =\}$, we reduce to T.

We now formally summarize what we have learned from the examples.

Conflicts – Case I

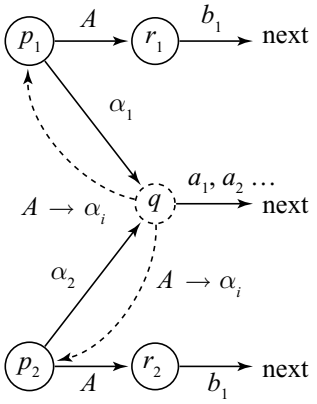
In the partial state diagram shown below, the state q has a Shift/Reduce conflict. If $FOLLOW(A) \cap \{a_1, a_2, \ldots\} = \varnothing$, then this conflict is resolvable using SLR(1) technique.



Here, $FOLLOW(p, A) = \{b\}$, $LA(p, A \rightarrow \alpha) = FOLLOW(p, A)$ and $LA(p, A \rightarrow \alpha) \cap \{a_1, a_2, \ldots\} = \varnothing$, and $FOLLOW(p, A)$ results in a read operation.

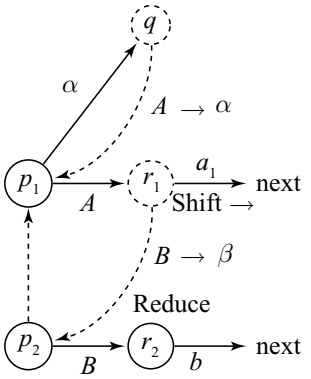
Conflicts – Case II

In the partial state diagram shown below, the state q has a Shift/Reduce conflict. Here, the state q can be accessed from a number of previous states – p_1, p_2, etc. The transitions are such that the item $[A \rightarrow \alpha_i \bullet]$ are in the set C_q. Here, $FOLLOW(p_1, A) = \{b_1\}$ and $FOLLOW(p_2, A) = \{b_2\}$. Then $LA(q, A \rightarrow \alpha_i) = \cup \{FOLLOW(p, A), \underset{\alpha}{p \Rightarrow} q\}$.



Conflicts – Case III

In the following partial state diagram, the state q has Shift/Reduce conflict. It is accessible from p_1 by string α. To resolve this Shift/Reduce conflict, we would need to know the $FOLLOW(p_1, A)$. When the state p_1 is uncovered, transition on A is to r_1, which also happens to be an inadequate state, with Shift/Reduce conflict.

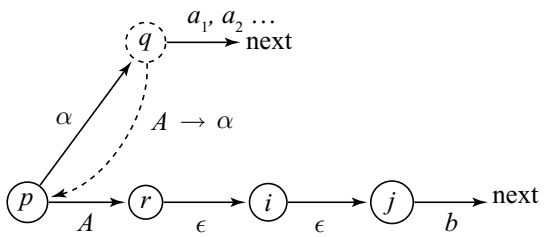


Then $FOLLOW(p_1, A)$ contains a_1 etc. plus $FOLLOW(p_2, B)$, etc.

Note that in a particular grammar Cases II and III may get combined.

Conflicts – Case IV

Generalization of Case I for ϵ productions.



4.3.11 Look-ahead (LA) Sets

For all LR(k) type parser construction, LR(0) is a good starting point. Each inadequate state q in the LR(0) finite-state machine for the grammar requires some look-ahead information for its resolution. Note that for any state with conflict, there should be at least one reduce operation associated with that state.

To find the look-ahead (LA) sets

$$\text{LA}(q,\ A \rightarrow \alpha) = \left\{ a \,\middle|\, a \in V_T,\ S \overset{+}{\underset{R}{\Rightarrow}} \delta A a \omega \text{ and } \delta\alpha \text{ accesses } q \right\}$$

Here, $\delta\alpha$ accesses the state q means that beginning with the start state of the FSM, scanning of the string $\delta\alpha$ will put the machine in state q.

Assuming that LR(0) item sets are already computed, we define

$$Reduce\ (q, a) = \left\{ A \rightarrow \alpha \,\middle|\, [A \rightarrow \alpha.] \in C_q \right\}$$

and $a \in V_T$. This means that *Reduce*(q, a) set contains the productions corresponding to all the completed items in set C_q.

4.3.12 LALR(1) Parser

A state q in LR(0) machine is inadequate if and only if $\exists a \in V_t$ such that $G(q, a)$ (a shift operation) is defined and $Reduce(q, a) \neq \varnothing$ or $|\ Reduce(q, a)\ | > 1$.

We have seen above various cases of conflicts and how a look-ahead can help resolve the conflict. In an LALR(1), systematically developed look-ahead checking techniques are used for such conflict resolution. We now discuss the detailed steps for computing the LA sets. Once these LA sets are known for each state in conflict, the actual parsing can proceed in a way similar to an SLR(1) parser.

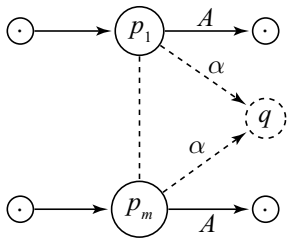
$$FOLLOW(p, A) = \left\{ a \,\middle|\, a \in V_T,\ S \overset{+}{\underset{\mathrm{R}}{\Rightarrow}} \delta A a \omega \text{ and } \delta \text{ accesses } q \right\}$$

This FOLLOW set contains those Terminal symbols that can immediately follow the Non-Terminal A in a sentential form, where prefix δ accesses state p. Note that this definition of the FOLLOW set already takes into account the state p where we want to find the FOLLOW set.

Then it can be shown that

$$LA(q, A \rightarrow \alpha) = \bigcup \left\{ FOLLOW(p, A) \,\middle|\, (p, A) \text{ is an NT transition and } p \overset{\alpha}{\Rightarrow} q \right\}$$

This means that the LA set for the rule $A \rightarrow \alpha$ at the state q is the set union of the FOLLOW sets for the transition from A from some other state p, which on reading α will arrive at state q.



When the rule $A \rightarrow \alpha$ is applied in state q, $|\alpha|$ symbols are popped off the stack, a state p_i is uncovered, which shifts A with a look-ahead symbol 'a'.

We define a useful relation

$$(q, A \rightarrow \alpha) LOOKBACK(p, A) \text{ iff } p \overset{\alpha}{\Rightarrow} q$$

$$\text{Then } LA(q, A \rightarrow \alpha) = \bigcup \left\{ FOLLOW(p, A) \mid (q, A \rightarrow \alpha)\ LOOKBACK(p, A) \right\}$$

Example

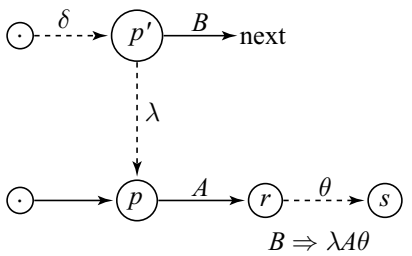
Using the same grammar as given in Section 4.3.10, the set of non-terminal transitions is {(0, E), (0, G), (0, T), (6, T), (8, T), (8, E)}.

The relation LOOKBACK is

{((2, $G \rightarrow f$), (0, G)), ((2, $T \rightarrow f$), (0, T)), ((4, $E \rightarrow T$), (0, E)), ((4, $E \rightarrow T$), (8, E)), ((7, $T \rightarrow f$), (6, T)), ((7, $T \rightarrow f$), (8, T)), ((9, $G \rightarrow E = E$), (0, G)), ((10, $E \rightarrow E + T$), (0, E)), ((10, $E \rightarrow E + T$), (8, E)), ((12, $T \rightarrow T * f$), (0, T)), ((12, $T \rightarrow T * f$), (6, T))}.

Also note that the FOLLOW sets are actually inter-related, i.e.

$$FOLLOW(p', B) \subseteq FOLLOW(p, A) \text{ if } B \rightarrow \lambda A\theta,\ \theta \underset{R}{\overset{*}{\Rightarrow}} \epsilon \text{ and } p' \overset{\lambda}{\Rightarrow} p$$



This can more easily be expressed in terms of a new relation:

$$(p, A)\ INCLUDES\ (p', B) \text{ iff } B \Rightarrow \lambda A\theta,\ \theta \underset{R}{\overset{*}{\Rightarrow}} \epsilon \text{ and } p' \overset{\lambda}{\Rightarrow} p$$

So $FOLLOW(p', B) \subseteq FOLLOW(p, A)$ if (p, A) $INCLUDES(p', B)$

For our example grammar, the INCLUDES relation is

$$((8, E), (0, G)), ((6, T), (0, E)), ((6, T), (8, E)), ((0, T), (0, E)), ((8, T), (8, E))$$

The reader should check the above relation against the SLR(0) item sets for the grammar and satisfy herself what this relation actually means.

We now define a set $Read(p, A)$ to contain all the Terminals $a \in V_T$ that are read before the phrases that contain A are reduced. The FOLLOW sets are computed using:

$$FOLLOW(p, A) = Read(p, A) \cup \bigcup \left\{ FOLLOW(p', B) \mid (p, A)\ INCLUDES(p', B) \right\}$$

Set $Read(p, A) = DR(p, A) \cup \bigcup \left\{ Read(t, C) \mid (p, A)\ READS(t, C) \right\}$

$$\text{where} \quad DR(p, A) = \left\{ a \in V_T \mid p \xrightarrow{A} t \xrightarrow{a} \right\} \text{ and}$$

$$(p, A) READS(t, C) \text{ iff } p \xrightarrow{A} t \xrightarrow{C}, \text{ and } C \overset{*}{\Rightarrow} \epsilon$$

For the example grammar, we can easily obtain from the LR(0) machine: $DR(0, E) = \{+, =\}$, $DR(0, T) = \{*\}$, $DR(0, G) = \{\#\}$, $DR(8, T) = \{*\}$, $DR(8, E) = \{+\}$, $DR(6, T) = \{*\}$.

Since we do not have any empty productions, $Read(p, A) = DR(p, A)$.

The FOLLOW sets can be calculated as

$$FOLLOW(0, G) = Read(0, G) = \{\#\}$$
$$FOLLOW(0, E) = Read(0, E) = \{+, =\}$$
$$FOLLOW(0, T) = Read(0, T) \cup FOLLOW(0, E) = \{*\} \cup \{+, =\} = \{*, +, =\}$$

etc.

We can now calculate the LA sets, using LOOKBACK functions, e.g.

$LA(2, G \rightarrow f) = FOLLOW(0, G) = \{\#\}$

$LA(2, T \rightarrow f) = FOLLOW(0, T) = \{*, +, =\}$

$LA(4, E \rightarrow T) = FOLLOW(0, E) \cup FOLLOW(8, E) = \{+, =\} \cup \{+, \#\} = \{+, =, \#\}$, etc. Proceeding in the same way:

$LA(7, T \rightarrow f) = \{*, +, =, \#\}$

$LA(9, G \rightarrow E = E) = \{\#\}$

$LA(10, E \rightarrow E + T) = \{+, =, \#\}$

$LA(12, T \rightarrow T * f) = \{*, +, =, \#\}$

Note that the Reduce/Reduce state 2 is resolved.

It is generally found that an LALR(1) grammar requires much less memory to store the parsing tables compared to LR(1) and that is why it is generally preferred. Figure 4.11 shows a hierarchy of context-free grammars. A grammar is said to be LALR(1) if its LALR(1) parsing table contains no conflicts. All SLR grammars are LALR(1), but not vice versa. Any reasonable programming language has an LALR(1) grammar, and there are many parser-generating tools available for LALR(1) grammars. For this reason, LALR(1) has become a standard for programming languages and for automatic parser generators.

In a real-life parser-generating project, if LALR(1) technique is to be used, these computations would be done using a computer, or a compiler building tool like `yacc` is used.

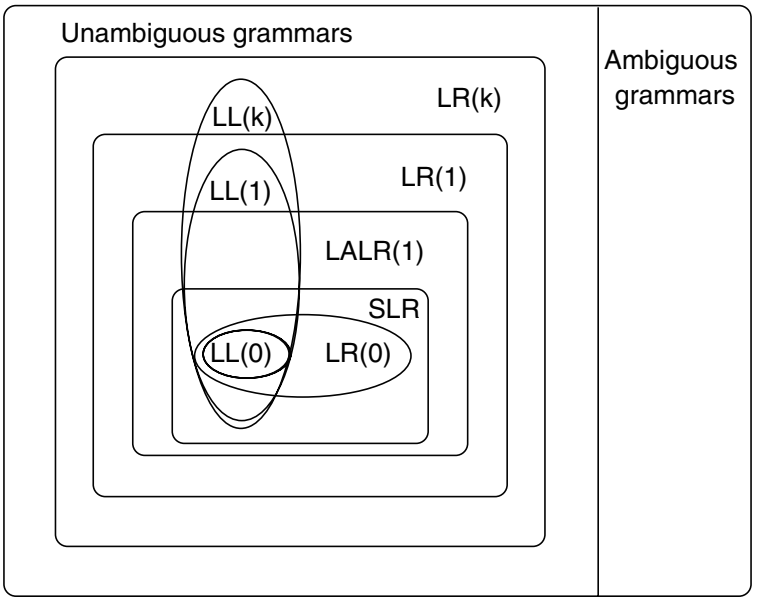


Fig. 4.11 A hierarchy of grammar classes

4.4 Yacc – A Parser Generator

A number of programming languages are developed and new ones will surely be developed in future. Existing languages are being enhanced or modified. New CPU architectures are being designed and implemented. For each new language or modification in a language and for each new or modified CPU architecture, we require a new compiler. This is not a trivial requirement and those responsible for providing such compilers are dreaming if the work of compiler writing can be automated, i.e. they want to have a compiler–compiler.

A compiler–compiler (CC) will, ideally speaking, generate a compiler $C_{L,M}$ for a given language L and given machine (i.e. CPU instruction set), including the optimization passes (see Fig. 4.12).

At present this ideal is not reached, i.e. no such CC is available, but rather several CCs are available approaching this ideal to various degrees.

Note that an ideal CC requires *two* distinct description languages:

- for describing the source language grammar, usually a modified BNF form;
- a hardware or instruction set processor (ISP) description language, by which we can specify the target language, the machine architecture (number of CPU registers, memory organization, addressing modes, etc.), in other words the *programming model* for that machine.

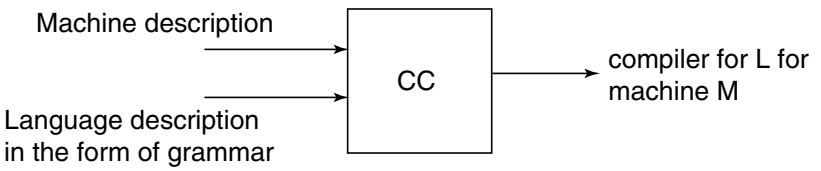


Fig. 4.12 A compiler–compiler (theoretical)

One such non-ideal CC is YACC. It is a "front-end" CC, i.e. it provides automatic generation of the initial phases of a full compiler, the Parser. A companion program, LEX, can be used to generate the lexical analyzer or the Scanner. The later phases, viz., semantic analysis, code-generation, storage allocation, symbol-table handling and optimization, etc., are to be added "manually" to make a complete and full compiler.

On GNU/Linux systems, an improved version, called `bison`, is available, which is upward compatible with YACC. A companion lexical analyzer generator named `flex` is also available. For further details, see the main pages for these programs.

4.4.1 YACC – A Compiler Writing Tool

The name YACC is an acronym from Yet Another Compiler Compiler. It was developed by S. C. Johnson in 1978 and was first implemented on UNIX operating system. At present it is also available on MS-DOS systems.

YACC accepts a grammar specified in a modified BNF and will do the following three major jobs:

1. Indicate if there are any problems with the grammar which make it unsuitable for building a language recognizer.
2. Generate a function in C language, named `yyparse()`, along with several tables to implement a (table-driven) parser for the specified language.
3. *Optionally* give several secondary outputs which help in debugging the grammar and make the job of writing a full compiler easy. It also provides several "hooks" by which other phases of the compiler can be made to communicate easily with the Parser.

While doing the above, YACC will take several, generally desirable, automatic actions, like resolution of conflicts, see Section 4.4.6.

The input file to YACC is generally given the file name extension .y and YACC generates up to three different files as output, depending upon the control switches used in the command line invoking YACC (see Fig. 4.13). For details of input format, see Section 4.4.3.

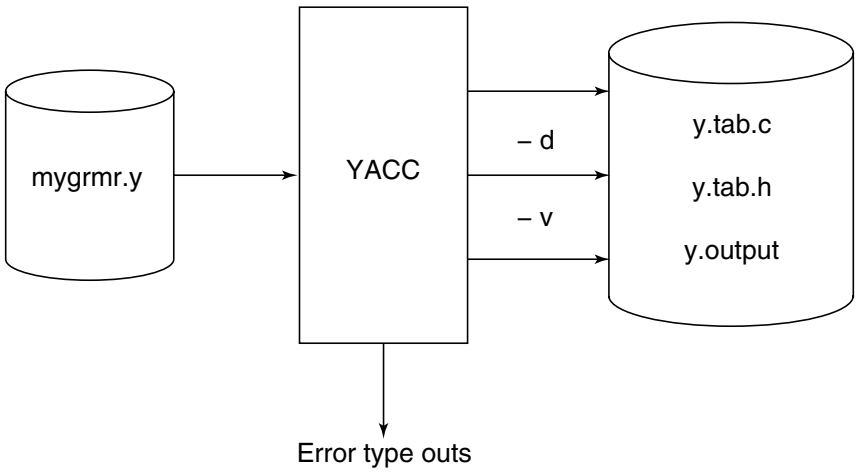


Fig. 4.13 YACC: options and outputs

Note: The file names are for Unix systems. For MS-DOS, the first '.' in file name is not used

Symbolic token names specified in the grammar are assigned integer indices by YACC and output as #define statements in the generated parser. As it is desirable that the same indices be used throughout the compiler, a separate file `y.tab.h` is created if we use `-d` switch in the command line.

If it found that that are serious errors in the grammar, another file named `y.output` may be optionally generated using the switch `-v`. This file in fact represents a detailed state diagram of the FSM part of the PDA implementing the parser.

Important components of the file `y.tab.c` are shown in Fig. 4.14. The parser function `yyparse()` calls the lexical analyzer function – *which is to be provided by the compiler writer* – `yylex()` to get the next token in the source program. It then interprets the tables depending upon the tokens provided by `yylex()` and modifies the stack contents. It also initiates the *Actions* specified for each of the language construct recognized. Several global variables like `yychar, yylval, yyval`, etc. are used by the parser for communicating with other functions.

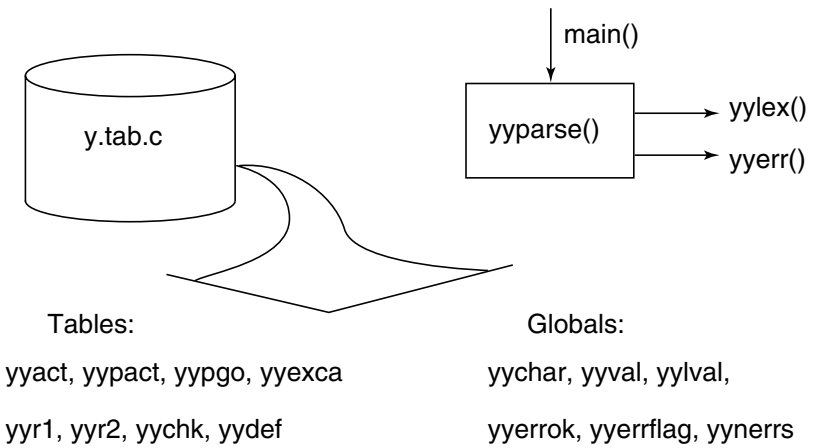


Fig. 4.14 Major components of `y.tab.c` file

4.4.2 Working of the Parser Generated by YACC

YACC generates a parser while analyzing the grammar. The parser is a push-down automaton (PDA) – a stack machine – where a large stack holds the current state. A transition matrix specifies a new state for each possible combination of the current state and the next input symbol. This and a table of user definable *Actions* form the other important components of the PDA. The interpreter within `yyparse()` repeatedly calls on a lexical analyzer function `yylex()` to get the next token. It returns a '0' to indicate that a valid sentence in the specified language was recognized (see Fig. 4.15).

Current state on the TOS and the next Terminal symbol supplied by `yylex()` select an operation from the transition matrix.

As an example of the first function of YACC, feed the grammar in file `example01.y`, (see Fig. 4.16), to YACC with the command:

```
yacc -v example01.y
```

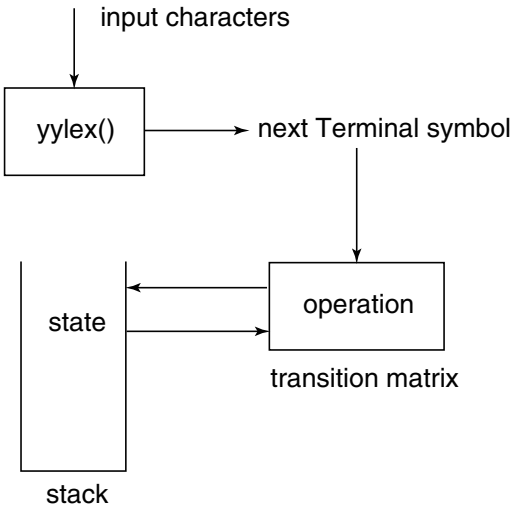


Fig. 4.15 Operation of `yyparse()`

```
%%
expression : expression '-' 'i'
           | 'i'
           ;
%%
```

Fig. 4.16 A very simple grammar: file `example01.y`

Nothing seems to have happened except that a file `y.output` is generated, which should be renamed as `example01.out`. It is suggested that you prepare a state-diagram of the machine represented by this file. A comparison of this state diagram with the original grammar shows that the parser indeed recognizes any sentence in the language generated by the grammar.

The actions specified for each grammar construct are converted into a switch statement towards the end of the parser. The parser seems to be large because it contains considerable amount of code for debugging and error recovery.

The transition matrix specifies five types of operations:

1. **accept:** Happens only once when `yylex()` returns a zero or negative value indicating end of the input sentence.
2. **error:** A wrong token for this particular state has been received.
3. **shift:** *new state*. This operation indicates that the next Terminal symbol is acceptable in the current state. The new state is pushed onto the stack and becomes the current state.
4. **goto:** *new state*. A reduce operation described below generates a Non-Terminal (NT) symbol to be used prior to the next Terminal symbol. Goto is the shift operation for the NT symbol.
5. **reduce:** *production number*. Pop as many states off the stack as the production has symbols on its RHS. The uncovered state on TOS is the new current state. The NT symbol whose production was just completed is used prior to the next Terminal symbol which in turn will be processed after the NT symbol.

4.4.3 Input Specification

In general, the input file to YACC should specify the grammar and actions in the following form:

```
declarations
%%
rules
%%
programs
```

The declarations and programs are optional. The rules section is made of one or more grammar rules in the form:

```
A : body;
```

where *A* represents a Non-Terminal and body represents a sequence of zero or more Non-Terminals and Terminals. The colon ':' and semicolon ';' are YACC punctuation. If there are several grammar rules with the same left-hand side (LHS), then a vertical bar '|' can be used to avoid rewriting the LHS. Thus, a rule with several alternative productions can be written as:

```
A : B C D
  | E F
  | G
  ;
```

The tokens should be declared by a %token construct. For example, the simple grammar of Fig. 4.16 will generate a language `{i, i-i, i-i-i, ...}`. If we want to allow more flexible syntactic type `IDENTIFIER` in place of literal '*i*', then the grammar should be as shown in Fig. 4.17.

```
%token IDENTIFIER
%%
expression  : expression '-' IDENTIFIER
            | IDENTIFIER
            ;
%%
```

Fig. 4.17 A very simple grammar: file `example02.y`

Note that the token value `IDENTIFIER` will be defined by YACC (to be 257 in this case) and the user supplied `yylex()` should return the same integer value when it detects a token `IDENTIFIER`. It is for this purpose that the `-d` switch is provided in YACC, i.e. by giving command:

```
yacc -d mygrmr.y
```

you will get both `y.tab.c` – the parser function plus tables, and `y.tab.h` – the header file containing the definitions of such tokens.

In order to see the action of YACC in debugging a grammar, give the command:

```
yacc -v example02.y
```

and you will get a file `y.output`, which you should rename to `example02.out`:

```
  0 $accept : expression $end

  1 expression : expression '-' IDENTIFIER
  2            | IDENTIFIER
state 0
$accept : . expression $end (0)

IDENTIFIER shift 1
. error

expression goto 2

state 1
expression : IDENTIFIER . (2)

. reduce 2

state 2
$accept : expression . $end (0)
expression : expression . '-' IDENTIFIER (1)

$end accept
'-' shift 3
. error

state 3
expression : expression '-' . IDENTIFIER (1)

IDENTIFIER shift 4
. error

state 4
expression : expression '-' IDENTIFIER . (1)

. reduce 1
4 Terminals, 2 nonterminals
3 grammar rules, 5 states
```

The `y.out` file on some systems also gives some statistics, which can be ignored in normal use. It may be useful when you are processing very large grammars.

The output is a representation of the FSM part of the parser-PDA and corresponds to the state diagram given in Fig. 4.18.

From the above, we see that:

- First of all, YACC adds a rule numbered 0:
  - `$accept : expression $end`

  in our example, and in general:
  - `$accept : <start symbol of grammar> $end`

  The place marker '.' is placed just before the *start symbol* and that gives us rule number 0.
- Then, following the rules of LR(0) item-set construction, various states are generated, with a few minor differences.

- The *shift* operation in the LR(0) item-set construction is replaced by two operations:
  - *shift* for a Terminal symbol or token,
  - *goto* for a Non-Terminal symbol.
- When the position marker is moved across the special symbol `$end`, which is a non-positive value from `yylex()`, *Accept* operation takes place, i.e. `yyparse()` returns a value 0.
- Any symbols other than those specified are indicated by '.'.

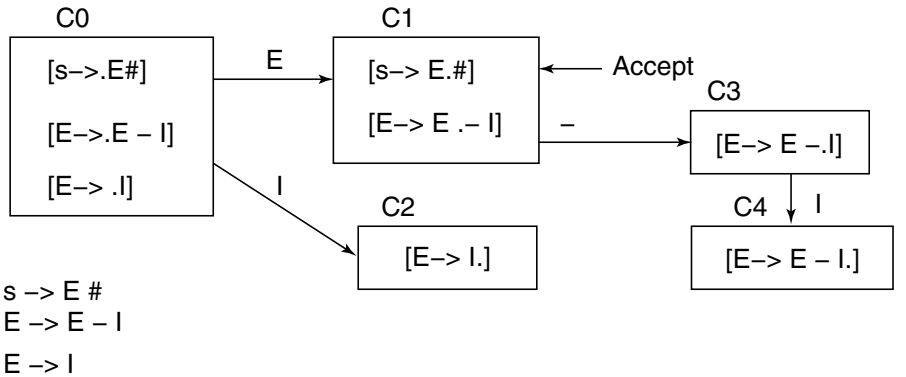


Fig. 4.18 Parser state diagram

4.4.4 Recursion in Grammar Specification

When specifying a list, we may do so using left recursion,

```
list:
     item
     | list ',' item
     ;
```

or right recursion:

```
list:
     item
     | item ',' list
```

If right recursion is used, all items on the list are first pushed on the stack. After the last item is pushed, `yacc` starts reducing. With left recursion, `yacc` never has more than three terms on the stack, since it reduces as it goes along. For this reason, it is advantageous to use left recursion.

4.4.5 Actions

The parser as created should not only decide if the given sentence is valid in the specified language, but also should generate outputs for the next phase of the compiler. YACC has special facilities for this purpose, known as *Actions*.

With each grammar rule the user may associate **actions** to be performed each time the rule is recognized. An action is an arbitrary C language statement. It is specified by one or more statements enclosed in the C block brackets { }.

The $ symbol is used as a special indicator. In order to return a value associated with the LHS symbol, a value is assigned to pseudo-variable $$. In order to obtain the values returned by the previous actions and the lexical analyzer (Scanner), the pseudo-variables $1, $2, etc. can be used, which correspond to the values returned by the respective component of the RHS of the rule, reading from left to right. Thus, if the rule is

```
A : B C D ;
```

then $2 has the value returned by C and $3 is the value returned by D.

Sometimes, it is desirable to specify an action before a rule is fully passed. YACC permits an action to be returned in the middle of a rule. For example, in the rule

```
A  :  B { $$ = 1;}
      C { x = $2; y = $3; }
   ;
```

the effect is to set *x* to 1 and *y* to the value returned by C. Note that the C blocks are also counted while indexing the pseudo-variables.

Let us consider the following example of a very preliminary four-function calculator (Fig. 4.19): `yyparse()` maintains a *value-stack* in parallel with the state-stack. Whenever the current state is pushed on the state-stack during a *shift* or *goto* operation, a value is also pushed on the value-stack.

```
/* Desk Calculator */
%token CONS
%left '+' '-'
%left '*' '/'
%%
line : /* empty */
     | line expr '\n' {printf("%d \n", $2);}
     ;
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | CONS { /* $$ = $1 by default */}
     ;
%%
```

Fig. 4.19 A four-function calculator: file `calculator.y`

The value to be pushed during a *shift*, i.e. acceptance of a Terminal symbol, is taken from the global variable `yylval`, which usually comes from the `yylex()` function.

In the above example, `yylex()` should set the value of `yylval` = the value of the constant, whenever it detects a constant and return the token value CONS to `yyparse()`.

The value to be pushed on the value-stack during a *goto* operation is taken from the global variable `yyval`. $$ represents `yylval`. Note that this happens during a reduction.

The data type of `yylval`, `yyval` and the value-stack, by default, is `integer`, but it can be set to any other type by, for example,

```
#define YYSTYPE double
```

This will allow use of `double`, i.e. double-precision floating-point numbers.

4.4.6 Ambiguity and Conflict Resolution

A grammar may have ambiguous rules such that it can do two legal operations, a shift or a reduction and has no way of deciding between them. This is called *Shift/Reduce conflict*. It is also possible that the parser has a choice of two possible and legal reductions. This is called a *Reduce/Reduce conflict*.

YACC uses two *disambiguating* rules whenever faced with such conflicts:

- In a Shift/Reduce conflict, the default is to do shift.
- In a Reduce/Reduce conflict, the default is to reduce by the earlier grammar rule.

Another way in which conflicts can be resolved is by using precedence rules. YACC allows us to specify the precedence of certain Terminal symbols, like mathematical operators, which are used for resolving the conflicts.

When YACC processes a grammar specification and finds conflicts, it reports the number of such conflicts. It is advisable to inspect the `y.output` file carefully to check if the resolutions attempted by YACC are acceptable or not.

For example, consider a grammar:

```
%token IFTHEN
%token ELSE
%%
statement  : ';' /* empty statement */
           | IFTHEN statement
           | IFTHEN statement ELSE statement
           ;
%%
```

This grammar results in Shift/Reduce conflict and is reported by YACC as such. The `y.output` file is

```
  0 $accept : statement $end

  1 statement : ';'
  2           | IFTHEN statement
  3           | IFTHEN statement ELSE statement

state 0
$accept : . statement $end (0)

IFTHEN shift 1
';' shift 2
. error

statement goto 3

state 1
statement : IFTHEN . statement (2)
statement : IFTHEN . statement ELSE statement (3)
```

```
IFTHEN shift 1
';' shift 2
. error

statement goto 4

state 2
statement : ';' . (1)

. reduce 1

state 3
$accept : statement . $end (0)

$end accept

4: Shift/Reduce conflict (shift 5, reduce 2) on ELSE
state 4
statement : IFTHEN statement . (2)
statement : IFTHEN statement . ELSE statement (3)

ELSE shift 5
$end reduce 2

state 5
statement : IFTHEN statement ELSE . statement (3)

IFTHEN shift 1
';' shift 2
. error

statement goto 6

state 6
statement : IFTHEN statement ELSE statement . (3)

. reduce 3

State 4 contains 1 Shift/Reduce conflict.

5 Terminals, 2 non-terminals
4 grammar rules, 7 states
```

YACC intentionally allows such conflicts and uses the rules mentioned above for their resolution.

As another example, consider the grammar:

```
%token IDENTIFIER
%%
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | IDENTIFIER
     ;
%%
```

If this grammar is input to YACC as such, it will result in:

yacc: 16 Shift/Reduce conflicts.

YACC allows such grammar, provided the precedence of the Terminals '+', '−', '*', '/' are defined. Then it will use it to resolve the conflict. The modified grammar is:

```
%token IDENTIFIER
%left '+' '-'
%left '*' '/'
%%
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | IDENTIFIER
     ;
%%
```

One may also use %right and/or %nonassoc.

4.4.7 Error Handling

YACC is usually used to develop a compiler and a compiler is almost always dealing with programs with errors in them. (How many times is a typical program compiled before it runs successfully?) Thus, error recovery is an important aspect of any compiler generator. Error handling is extremely difficult area because many of the errors are semantic in nature. In order to handle errors fully, several complicated actions are to be taken.

If no particular steps are taken by the user, the parser generated by YACC will simply reject the source program on the very first encountered error, without checking the remaining part of the program. This is not very helpful. What is desirable is that the parser should recover sufficiently from the error to check the remaining part of the program, starting with a point as close to the error as possible.

YACC provides a special token named `error`, which can be used in grammar rules and suggests places where errors are expected and recovery is possible. The parser pops its stacks till it enters a state where the token `error` is legal. It then behaves as if `error` is the current look-ahead token and performs the action specified. For example,

```
stmt:
        ';'
| expr ';'
| PRINT expr ';'
| VARIABLE '=' expr ';
| WHILE '(' expr ')' stmt
| IF '(' expr ')' stmt %prec IFX
| IF '(' expr ')' stmt ELSE stmt
| '{' stmt_list '}'
| error ';'
| error '}'
;
```

The `error` token is a special feature of `yacc` that will match all input until the token following error is found. For this example, when `yacc` detects an error in a statement it will call `yyerror`,

flush input up to the next semicolon or brace, and resume scanning. In order to prevent a cascade of error messages, the parser, after detecting an error, remains in an error state until three tokens have been successfully read and shifted.

4.4.8 Arbitrary Value Types

The values returned by the actions and the lexical analyzer are integers by default. YACC can support values of other types, including C structures. The value-stack is declared to be a `union` of the various types of the values desired. The user declares the union and associates union member names with each token and Non-Terminal having a value. When the value is referenced through the $ construct, YACC will automatically insert the appropriate union name.

To declare the union the user includes in the declaration section:

```
%union {
          <body of the union>
}
```

which declares the YACC value-stack. The external variables `yylval` and `yyval` will have this union type.

4.4.9 Debugging `yacc`

Yacc has facilities that enable debugging. This feature may vary with different versions of yacc, so you should consult the documentation for the version you are using for the details. The code generated by `yacc` in the file `y.tab.c` includes debugging statements that are enabled by defining `YYDEBUG` and setting it to a non-zero value. This may also be done by specifying command-line option `-t`. With YYDEBUG properly set, debug output may be toggled on and off by setting `yydebug`. Output includes tokens scanned and Shift/Reduce actions.

```
%{
#define YYDEBUG 1
%}
%%
...
%%
int main(void) {
   #if YYDEBUG
      yydebug = 1;
   #endif
   yylex();
}
```

In addition, you can dump the parse states by specifying command-line option `-v`. States are dumped to file `y.output` and are often useful when debugging a grammar. Alternatively, you can write your own debug code by defining a TRACE macro, as illustrated below. When DEBUG is defined, a trace of reductions, by line number, is displayed.

```
%{
#ifdef DEBUG
#define TRACE printf("reduce at line %d\n", __LINE__);
#else
```

```
#define TRACE
#endif
%}
%%
statement_list:
                statement
                    { TRACE $$ = $1; }
              |  statement_list statement
                    { TRACE $$ = newNode (';', 2, $1, $2); }
              ;
```

4.4.10 Working Examples

We round up this discussion with two examples:

A Very Simple Grammar

See the grammar in Fig. 4.17. This grammar does not have any actions specified, so we do not expect any outputs. It is only an *Acceptor*, i.e. will only say if the given sentence is valid. The lexical analyzer is

```
#include<stdio.h>
#include<ctype.h>
#include "defs.h"

char *progname;
int lineno = 1;

main(int argc, char *argv[])
{
  progname = argv[0];
  yyparse();
}

yylex()
{
  int c;
  char str[20];

  while((c = getchar()) == ' ' || c == '\t')

  if(c == EOF)
    return 0;
  if(isalpha(c)){
  ungetc(c, stdin);
  scanf("%s", str);
  return IDENTIFIER;
}
  if(c == '\n'){
    lineno++;
    return 0;
```

```
  }
  return c;
}

yyerror(s)
     char *s;
{
  warning(s, (char *) 0);
}

warning(s, t)
     char *s, *t;
{
  fprintf(stderr, "%s: %s", progname, s);

  if(t)
    fprintf(stderr, "%s", t);
  fprintf(stderr, "near line %d\n", lineno);

}
```

The `defs.h` file contains links to the files generated by `-d` switches while processing `example02.y` and `calculator.y`:

```
#include "example02.h"
#include "calculator.h"
```

This program should be compiled with command:

```
yacc -d example02.y
mv y.tab.c example02.c
mv y.tab.h example02.h
yacc -d calculator.y
mv y.tab.c calculator.c
mv y.tab.h calculator.h
           gcc -o example02 util.c example02.c
```

Calculator

This is a four-function integer calculator, as given by grammar in Fig. 4.19. It requires a slightly different `yylex()`, as given in the file `util1.c`. (Why is this so?):

```
yylex()
{
  int c;
  while((c = getchar()) == ' ' || c == '\t');
  if(c == EOF)
    return 0;
  if(isdigit(c))
    {
      ungetc(c, stdin);
      scanf("%d", &yylval);
```

```
      return CONS;
    }
  if(c == '\n')
    lineno++;
  return c;
}
```

Compile this program using a command:

```
gcc -o calculator util1.c calculator.c
```

In anticipation of our further discussion about `yacc` in the subsequent chapters, you are strongly advised to work through Chapter 8, "Program Development" in the book "The UNIX Programming Environment" by Kernighan and Pike.

4.5 Other Parser Generators

4.5.1 Bison

Bison is a parser generator by GNU organization, similar to `yacc`, written mainly by Robert Corbett and Richard Stallman. It is generally upward compatible with input files designed for `yacc`.

Input files should follow the `yacc` convention of ending in `.y`. Unlike `yacc`, the generated files do not have fixed names, but instead use the prefix of the input file. Moreover, if you need to put C++ code in the input file, you can end the name by a C++-like extension (`.ypp` or `.y++`), then bison will follow your extension to name the output file (`.cpp` or `.c++`). For instance, a grammar description file named `parse .yxx` would produce the generated parser in a file named `parse.tab.cxx`, instead of `yacc's y.tab.c`.

Bison has several options to control the types of outputs, including a graph of the FSM. The latest version (2.4.3 – 05-Aug-2010) has several interesting extensions beyond the original `yacc`. We describe here some of them.

General LR Parse – GLR

This seems to be the most interesting extension to `yacc`. As we know `yacc` is a generator for an LALR(1) parser. It can also handle in some cases look-ahead by 2 and implement partial LR(1) behaviour. GNU Bison adds a new capability – "parallel" search of reductions to resolve conflicts. We quote from the Bison documentation:

> In some grammars, Bison's standard LALR(1) parsing algorithm cannot decide whether to apply a certain grammar rule at a given point. That is, it may not be able to decide (on the basis of the input read so far) which of two possible reductions (applications of a grammar rule) applies, or whether to apply a reduction or read more of the input and apply a reduction later in the input. These are known respectively as Reduce/Reduce conflicts and Shift/Reduce conflicts.
>
> To use a grammar that is not easily modified to be LALR(1), a more general parsing algorithm is sometimes necessary. If you include `%glr-parser` among the Bison declarations in your file, the result is a generalized LR (GLR) parser. These parsers handle Bison grammars that contain no unresolved conflicts (i.e. after applying precedence declarations) identically to LALR(1) parsers. However, when faced with unresolved Shift/Reduce and Reduce/Reduce conflicts, GLR parsers use the simple expedient of doing both, effectively cloning the parser to follow both possibilities. Each of the resulting parsers can again split, so that at any given time, there

can be any number of possible parses being explored. The parsers proceed in lockstep; that is, all of them consume (shift) a given input symbol before any of them proceed to the next. Each of the cloned parsers eventually meets one of two possible fates: either it runs into a parsing error, in which case it simply vanishes, or it merges with another parser, because the two of them have reduced the input to an identical set of symbols.

During the time that there are multiple parsers, semantic actions are recorded, but not performed. When a parser disappears, its recorded semantic actions disappear as well and are never performed. When a reduction makes two parsers identical, causing them to merge, Bison records both sets of semantic actions. Whenever the last two parsers merge, reverting to the single-parser case, Bison resolves all the outstanding actions either by precedences given to the grammar rules involved or by performing both actions, and then calling a designated user-defined function on the resulting values to produce an arbitrary merged result.

Parser Written in C++ and Java

Bison can generate parsers written in C++ and Java, with a few limitations at present.

4.5.2 Parse::Yapp

As Perl language has powerful string manipulation and regular expression handling facilities, it is a good candidate for implementing a Parser generator. One of the more well-known ones is `Parse::Yapp`, which is a Perl extension for generating and using LALR parsers. Parse::Yapp (Yet Another Perl Parser compiler) is a collection of modules that let you generate and use yacc-like thread safe (re-entrant) parsers with Perl object-oriented interface.

The script `yapp` is a front-end to the `Parse::Yapp` module and lets you easily create a Perl object-oriented parser from an input grammar file.

The grammar file for Yapp has a syntax similar to the grammar file for `yacc`.

4.5.3 ANTLR

Another Tool for Language Recognition

ANTLR is the name of 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 mentor is Professor Terence Parr of the University of San Francisco.

ANTLR rules are expressed in a format deliberately similar to EBNF instead of the regular expression syntax employed by other parser generators.

At the moment, ANTLR supports generating code in the following languages: C, C++, Java, Python, C#, Objective-C. ANTLR-3 is a free software, published under a three-clause BSD license.

For an introduction, see the ANTLR tutorial at the University of Birmingham. For background on the theory, see articles from the ANTLR pages, e.g. an ANTLR journal paper.

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 and -3 plug-ins for Eclipse hosted on Source-Forge.

4.6 Grammar for `miniC`

In Chapter 12, we shall discuss in detail, the development of a compiler for `miniC` language, which is a simplified version of C. We give here a slightly trimmed version of the `yacc` grammar for that

language. This grammar is an extended and modified version of grammar for `hoc6` put in public domain by AT&T. The copyright of the original `hoc6` is held by AT&T.

```
%union {
Symbol *sym; /* symbol table pointer */
util inst; /* machine instruction */
int narg; /* number of arguments */
}
%token <sym>NUMBER STRING PRINT VAR BLTIN UNDEF WHILE FOR IF ELSE
%token <sym>FUNCTION PROCEDURE RETURN FUNC PROC READ INT IVAR SVAR END
%token <narg>ARG
%type <sym> expr
%type <inst> stmt asgn prlist stmtlist
%type <inst>cond while for if begin end
%type <sym>procname
%type <narg>arglist
%type <sym> dtype
%type <sym> dlist
%right '=' ADDEQ SUBEQ MULEQ DIVEQ MODEQ
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/' '%'
%left UNARYMINUS NOT INC DEC
%right '^'
%%
list:  {preproc();} stmtlist END { postproc(1); return 1;}
       | {preproc();} stmtlist error END { yyerrok; postproc(1);return 1;}
;
asgn:  VAR '=' expr        { }
       | IVAR '=' expr     { }
| VAR ADDEQ expr { }
| VAR SUBEQ expr { }
| VAR MULEQ expr { }
| VAR DIVEQ expr { }
| VAR MODEQ expr { }
| ARG '='   expr { }
| ARG ADDEQ expr { }
| ARG SUBEQ expr { }
| ARG MULEQ expr { }
| ARG DIVEQ expr { }
| ARG MODEQ expr { }
;
stmt: decl
        | expr { }
| RETURN { }
| RETURN expr { }
| PROCEDURE begin '(' arglist ')' { }
| PRINT prlist { }
```

```
| while'(' cond ')' stmt end { }
| for'(' cond ';' cond ';' cond ')' stmt end { }
| if'(' cond ')' stmt end { }
| if'(' cond ')' stmt end ELSE stmt end { }
| '{' stmtlist '}'{ }
;
cond:    expr { }
;
while:   WHILE { }
;
for:     FOR { }
;
if:      IF { }
;
begin:   /* nothing */ { }
;
end:     /* nothing */ { }
;
stmtlist: /* nothing */ { }
| stmtlist '\n'
| stmtlist stmt
;
expr:    INT { }
         | NUMBER { }
| VAR    { }
| IVAR   { }
| ARG    { }
| asgn
| FUNCTION begin '(' arglist ')' { }
| READ '(' VAR ')' { }
| BLTIN '(' expr ')' { }
| '(' expr ')'{ }
| expr '+' expr  { }
| expr '-' expr  { }
| expr '*' expr  { }
| expr '/' expr  { }
| expr '%' expr  { }
| expr '^' expr  { }
| '-' expr %prec UNARYMINUS { }
| expr GT expr   { }
| expr GE expr   { }
| expr LT expr   { }
| expr LE expr   { }
| expr EQ expr   { }
| expr NE expr   { }
| expr AND expr  { }
| expr OR expr { }
| NOT expr { }
```

```
| INC VAR { }
| DEC VAR { }
| VAR INC { }
| VAR DEC { }
;
prlist: expr { }
| STRING { }
| prlist ',' expr { }
| prlist ',' STRING { }
;
defn: FUNC procname { }
    '(' ')' stmt { }
| PROC procname { }
    '(' ')' stmt { }
;
procname: VAR
| FUNCTION
| PROCEDURE
;
arglist: /* nothing */ { }
| expr { }
| arglist ',' expr { }
;
decl: dtype ':' dlist
        ;
dtype:    IVAR
        | VAR
        | SVAR
        ;
dlist:    VAR             { }
        | dlist ',' VAR   { }
        ;
%%
```

4.7 Symbol Table and Parser

We already had a preliminary discussion about Symbol Tables in Chapter 3 (Section 3.2). A Symbol Table exists throughout the compilation steps, but it has a rather limited role in the Parser. Basically, it accesses the Symbol Table to ensure that an operand exists. There are certain identifiers of which the Parser should be aware of, such as keywords, built-in functions and pre-defined constants and variables. They are often stored in the symbol table before the compilation process begins, so that Scanner and Parser are able to trap them. Also, user-defined declarations of variable and functions should be entered in the Symbol Table by the Parser for later processing by the semantic phase – like type checking, argument counts of the functions etc. We shall present here the additional functions needed to implement some of these facilities.

4.7.1 Pre-defined Entities

The simplest operation is insertion of pre-defined entities like keywords, built-in functions and pre-defined constants and variables. This is done through `init()` function given below.

```
/* install constants and built-ins in table */
void init(void){
     int i;
     Symbol *s;
     for(i = 0;  keywords[i].name; i++)
                 install(keywords[i].name, keywords[i].kval, 0.0);
     for(i = 0;  consts[i].name; i++)
                 install(consts[i].name, VAR, consts[i].cval);
     for(i = 0;  builtins[i].name; i++) {
                 s = install(builtins[i].name, BLTIN, 0.0);
                 ptr(s) = builtins[i].func;
     }
}
```

This `init()` function is invoked by the Parser at its beginning. The specification of keywords etc. is made available as arrays of structures as shown below:

```
struct{           /* Keywords */
       char   *name;
       int    kval;
} keywords[] = {
       "if",      IF,
       "else",    ELSE,
       "while",   WHILE,
       "for",     FOR,
       "int",     IVAR,
       "float",   VAR,
       "string",  SVAR,
       0,         0,
};
```

The pre-defined constants are stored as

```
struct{           /* Constants */
       char     *name;
       double   cval;
} consts[] = {
       "PI",     3.14159265358979323846,
       "E",      2.71828182845904523536,
       "DEG",   57.29577951308232087680, /* deg/radian */
       0,        0
};
```

The built-in functions are, for example,

```
static struct{      /* Built-ins */
       char *name;
       double(*func)(double);
```

```
} builtins[] = {
     "sin", sin,
     "cos", cos,
     0,     0
};
```

We have decided to include the built-in functions in the parser initialization, in anticipation of our presentation of a full compiler for a miniC language in Chapter 12. In the case of C, a large number of various function libraries are available and the C parser is made aware of the functions defined therein via the `#include'd` *Header* files.

Insertion of variable declarations, like `int:a,b`, involves semantic action of detecting and inserting the `type` of the variable and is explained in Chapter 5 (Section 5.4.7). The relevant portion of the grammar is:

```
%token  <sym> NUMBER STRING PRINT VAR BLTIN UNDEF WHILE FOR IF ELSE
%token  <sym> FUNCTION PROCEDURE RETURN FUNC PROC READ INT IVAR SVAR
%type   <sym> dtype
%type   <sym> dlist
decl:   dtype ':' dlist ;
dtype:  IVAR | VAR | SVAR ;
dlist:  VAR                { type($1) = type($0); }
      | dlist ',' VAR      { type($3) = type($0); }
      ;
```

Inserting the details of user-defined functions (and procedures) is slightly involved, as not only the return type of the function, but number and types of the formal arguments are to be handled. We postpone discussion about it till Chapter 12.

We tested the Scanner, with the above functions included, but without invoking the parser and we obtained the following result for inputs:

```
int : a

(275) 275
(275) 58
(263) 261
(263) 10

float:f

(261) 261
(261) 58
(263) 261
(263) 10

string:s

(276) 276
(276) 58
(263) 261
(263) 10
```

A dump of the Symbol Table gave something like this:

```
[DEG](261){42652ee1}
[E](261){402df854}
[PI](261){40490fdb}
[a](263){0}
[cos](262){8048ab8}
[else](267){0}
[f](263){0}
[float](261){0}
[for](265){0}
[if](266){0}
[int](275){0}
[s](263){0}
[sin](262){8048b98}
[string](276){0}
[while](264){0}
```

Notice that the variables `a`, `f` and `s` are shown with type as `UNDEF` and not their expected types IVAR (275), VAR (261) and SVAR (276), respectively. This has happened because we had suppressed the semantic phase and the inherited attributes transmission did not take place. Compare these results with those in Chapter 5 (Section 5.5).

4.8 Real-life – GCC: GNU Compiler Collection

GNU Compiler Collection (GCC) is a code development environment supporting several front-ends, like processors for C, C++, Objective-C, Fortran, Pascal, Ada, Java and others. Hence, it is called a compiler collection. It also provides several machines as targets – Intel, ARM, PowerPC, Motorola, Sparc, among others. It is an optimizing compiler, with highly modular design to encourage easy porting to new target machines, and easy addition of new languages (see Fig. 4.20). At this stage, you may not realize the significance of all the stages indicated in the figure, but note that the Scanner and the Parser for each language is contained in the *Front-end* for that language. The *Mid-end* is mainly concerned with code optimization and the *Back-end* is concerned with Code Generation. The output of GCC is an assembly language code, which is processed by a standard assembler. GCC is a Free Software Foundation Project and is offered under GPL licence. Suppose you want to implement a compiler for a new language X, why would you write a front-end for GCC rather than writing a full compiler for X, compiling directly to a target language? As a number of target machines are already present in GCC, considerable portion of your work is reduced. Also, all the optimizations in GCC are available, and better debugging information is generated. What you have to develop is the front-end for X, which is comparatively easier task, especially in view of the availability of tools like `yacc` and `lex`.

C, C++ and other programs are internally represented in GCC as *trees*. Thus, the central data structure used is a tree. The nodes of such trees are pointer type, the object pointed maybe of different types. All syntactical object types have corresponding tree nodes. Similarly, all source-level statement constructs have corresponding tree nodes, for example:

DECL_STMT – local declarations

DO_STMT – represents a do loop

IF_STMT – represents an IF statement

etc. Furthermore, trees can have attributes.

We shall discuss the other parts of the GCC at appropriate places throughout the rest of the chapters.

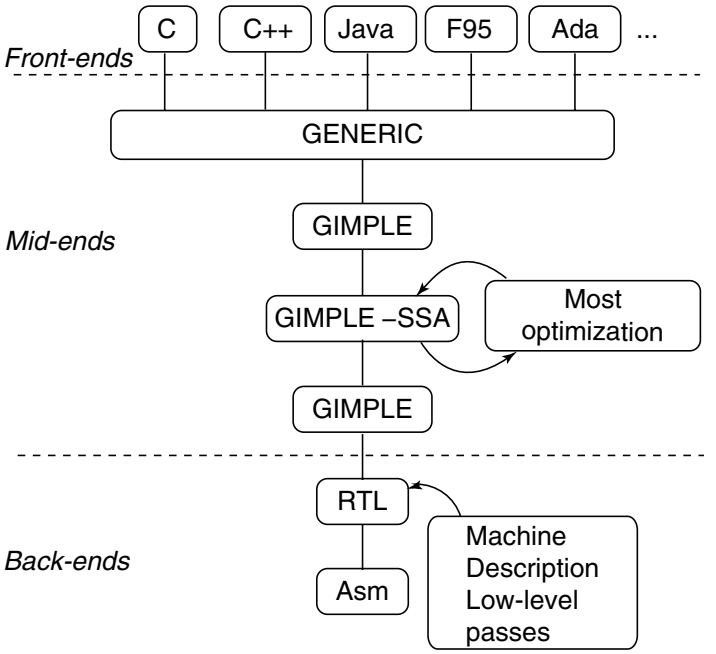


Fig. 4.20 Overall structure of GCC – GNU compiler collection

? EXERCISES

1. Check which of the following are LL(1):

```
(a)  S -> A | B        (b) S -> AB        (c)  S -> aAaB | bAbB
     A -> aA | a           A -> Ba             A -> S | cb
     B -> bB | b           B -> Cb | c         B -> cB | a
                           C -> c
```

2. Have you wondered why we did not discuss LL(k), $k > 1$ parsers, say LL(2) parsers? Try to form a parsing function $M()$ for LL(2) parser.
3. Try to convert the following grammar to a regular expression:

```
S -> aA    B -> bC
A -> aA    C -> cB
A -> aB    C -> c
```

4. Show that the following grammar is *not* LL(1):

```
V -> S#              A -> abS | c
S -> aAa | empty
```

5. Give the rules by which you can distinguish the following types of grammars: Regular, LL(1) with and without empty rules, context-sensitive, LR(0) and SLR(1).
6. Check if the following grammar is SLR(1) or not:

```
S -> E#        T -> empty
E -> T         T -> Ta
E -> E;T
```

7. What is the difference between the terms "viable prefix" and "valid prefix"?
8. Suppose for a language there are two distinct grammars. While parsing sentences in that language by parsers based on the two grammars, will the definition of a *handle* be the same or will it be different in the two cases?
9. Why is the following grammar not an SLR(1)?

```
S -> E#      E -> bEa | aEb | ba
```

10. State true/false: In a grammar being handled by YACC, Terminal symbols are recognized by `yylex()` and Non-Terminals by `yyparse()`.
11. Two values may in general be returned by `yylex()` to `yyparse()`, one is the return value of the function itself, and another, if present, is returned via _____.
12. YACC uses certain *disambiguating* rules while generating a parser for an ambiguous grammar. In a typical procedural language, in

```
if <cond1> then if <cond2> then S1 else S2
```

the `else` part is associated with second `if`. YACC ensures such a parsing by using the rule: (select most appropriate answer)

(a) in case of Shift/Reduce conflict, give precedence to shift;
(b) in case of Reduce/Reduce conflict, use the rule which is earlier in the grammar specification;
(c) use operator precedence and associativity;
(d) none of the above.

13. Develop a grammar in `yacc` syntax for the following "flow-chart" language F1 and implement a parser using `yacc`.

```
program : 'read' var+ ';' basicblock+
basicblock : label ':' assignment* jump
assignment : var '=' expr
jump : 'goto' label | 'if' expr 'goto' label 'else' label | 'return' expr
expr : constant | op expr expr
label : identifier | number
```

What should be the target language to actually draw the flow-charts?

14. Develop a grammar in `yacc` syntax for the following "Parentheses Language":

```
S -> L
L -> L P
L -> P
P -> ( P )
P -> ( )
```

Refer to this grammar as G_p. Input your grammar to yacc and obtain the LR(0) machine, after removing conflicts if any.

15. We plan to develop a Boolean expression calculator (BEC) for manipulating Boolean expressions. It is expected to handle Boolean expressions in "Sum-of-Products" form. The basic grammar is:

```
A -> V = E
E -> V | T | A
E -> E + E
E -> E * E
E -> E ^ E
E -> ( E )
E -> ! E
E -> E ? T
```

Here '+' denotes Boolean OR, '*' AND, '∧' EXOR, and '!' negate. *T* is a min-term expressed as a decimal number. Develop complete BEC grammar input for `yacc`, including assignment of a Boolean expression value to a variable. Test it on `yacc` for ambiguities and other problems. Refer to this grammar as G_b.

WEB RESOURCES

1. LL(*) Parsing: Prof. Terence Parr, University of San Francisco `http://www.antlr.org/workshop/ANTL`
2. The Compiler Generator Coco/R: University of Linz `http://www.ssw.uni-linz.ac.at/coco/`.

FURTHER READING

A standard reference text for Scanners and parsers is Aho, Sethi and Ullman [ASU86]. Advanced material is found in Muchnick [Muc97]. Detailed compiler design with C language implementation is discussed in Holub [Hol90]. Kernighan and Pike [KP87] have illustrated the use of `yacc` and `lex` in detail while describing development of various versions of Higher Order Calculator or HOC.

Complete details about the GNU Compiler Collection are available in two e-books – "Using the GNU Compiler Collection" and "GNU Compiler Collection Internals" both by Richard M. Stallman and the GCC Developer Community. These e-books are available at the GCC website: `http://gcc.gnu.org`. There are several research groups which have adopted GCC as a Vehicle for Compilation/Architecture Research, for example, see `http://www.hipeac.net`, "High-performance Embedded Architectures and Compilers".

GLOSSARY

Bookkeeping Keeping track of status of a system.
Buggy A program or a system with errors.
Compiler–compiler A program which, given specification of a language in terms of its grammar, generates major portions of its compiler modules. Usually the parser is generated.
Conflicts Alternate paths in the FSM of the parser model, which cannot be decided unless additional information is supplied.
Dangling Undecided.
Deterministic FSM is one in which each arc in the state diagram is associated with a well-defined set of input symbols, specifically there are no e transitions.

Disambiguation Resolution of ambiguities.
Double-precision refers to a computer representation of real numbers as floating-point numbers in mantissa/exponent form, where mantissa size is 53-bits signed 2's complement and exponent is 11-bits biased unsigned integer.
GLR-parser Generalized LR parser, which explores in parallel with the alternate paths in the FSM of the parser in order to achieve higher speed.
Item set A set of items, specifically corresponding to a state in the parser FSM.
Left-production is generated by a grammar rule by appending symbols on the LHS of the sentential form.
Look-ahead is a symbol that would be scanned next at the current stage of parsing, but is actually not yet read.
Min-term is a Boolean term obtained by AND of *all* the variables – either in direct or in negated form – making up a Boolean expression.
Nullable-NT is a non-terminal meta-variable in a grammar, which can be replaced by a null symbol by some grammar rule.
Parse-stack works along with the FSM of the parser in the PDA model.
Parser-PDA See parse-stack.
Reduce-reduce Conflict occurs when two different reduce operations are indicated within the same item set.
Reduction Replacing a *handle* by a Non-Terminal on the parse-stack, following a particular grammar rule.
Re-entrant Routine allows its second and subsequent invocations even when the first invocation is not completed, without any interference between the active invocations.
Right-context Symbols on the RHS of the scanning point, which provide a context for the parsing decisions, especially for conflict resolution.
Right-production is generated by a grammar rule by appending symbols on the RHS of the sentential form.
Single-precision Refers to a computer representation of real numbers as floating-point numbers in mantissa/exponent form, where mantissa size is 24-bits signed 2's complement and exponent is 8-bits biased unsigned integer.
State-stack is possibly a separate stack in the PDA of the parser, where the states of the FSM are saved as the parsing proceeds.
Sum-of-products form of Boolean expression is formed by AND-terms being OR'ed together, for example $x'y + xy'$.
Table-driven parser is one in which the state transition information is stored in one or more tables. This allows the basic parser program to be used with any valid grammar, just by changing the tables.
Uncover A state occurs when a reduction takes place and a handle is replaced by a Non-Terminal.
Value-stack is possibly a separate stack in the PDA of the parser, where the attribute values are saved as the parsing proceeds.

5
Syntax-directed Translation

What you will learn in this chapter

- What do we mean by "meaning" of a program?
- Semantics of a program
- Interplay of Syntax and Semantics
- What is "Syntax-directed Translation"?
- Static and Dynamic semantic analysis
- Methods of semantic specifications
- What is a Semantic stack and how it is used?
- Semantic stacks in `yacc`
- Actions in `yacc`
- What are Attribute grammars?
- What is a dependency graph?
- What are Inherited and Synthesized Attributes?
- What are S-type Definitions and L-type Definitions?
- Synthesized and Inherited Attributes in `yacc`
- Symbol Table handling and semantic analysis
- Symbol Table in `miniC`
- Intermediate Representation output for `miniC`

Key Words

semantic analysis, syntax-directed translation, semantic specifications, Semantic stack, actions, attribute grammars, inherited and synthesized attributes, Intermediate representation

Until now we have discussed how a program in a High Level language is analyzed, by Scanner and Parser, to find its grammatical structure and validate it as a grammatically correct program. However, this is not sufficient from the viewpoint of both achieving the desired translation to the target code and validating the semantic correctness of the program. Even in day-to-day English, both the sentences "cow eats grass" and "grass eats cow" are grammatically correct, but one of them is semantically, i.e. by its meaning, not correct.

Moreover, in the case of programming languages, a particular statement can have different shades of meaning depending upon how such a statement was meant to be interpreted in a language. For example, consider a statement `c = a + 5`.

No.	Type	Semantics	Formal Lang.	Chomsky
1	Declarative	Value of *c* equals value of *a* plus value 5	Regular, CFL	3,2
2	Algorithmic	To get value of *c* add 5 to value of *a*	CFL, CSL	2,1
3	Imperative	Add 5 to value of *a* and assign the result to *c*	CSL, PSL	1,0

The declarative interpretation simply specifies a relationship between '*c*' and '*a*'; in fact it can be used to find value of '*a*' given the value of '*c*' and vice versa. The algorithmic interpretation announces the method of finding the value of '*c*', if the value of '*a*' is known, but it does not specify who (which agent) will do that work and how will it be done. Of course, it is more specific that the value of '*c*' is to be obtained from the value of '*a*'. The imperative interpretation commands some agent (may be a CPU) to add number 5 to the value of '*a*' and *assign* the result to '*c*'.

This example shows that the same sequence of symbols may represent different meaning depending upon in which type of language it is written. This is the reason why it is necessary to specify, for each syntactic construct, what exact action the target machine is required to take. As the target machine will be controlled by the target language program, the process of finding out the meaning or semantics of the program should be completed before the target code can be generated. Thus apart from the grammar to specify the syntax, we need to have specification of the semantics of the language.

Consider an arithmetic expression 2+3. If we give this expression as an input to an *expression calculator*, a *type checker* and an *infix-to-prefix parser,* respectively, we get;

$2+3 \rightarrow \boxed{\text{calculator}} \rightarrow 5$

$2+3 \rightarrow \boxed{\text{type checker}} \rightarrow$ "integer"

$2+3 \rightarrow \boxed{\text{infix-to-prefix parser}} \rightarrow +2\ 3$

This example shows that different phases of a compiler may interpret a program segment in different ways, depending upon the analyses or synthesis they are doing.

Going back to the two English sentences "cow eats grass" and "grass eats cow", note that we are able to say that the second sentence is wrong in English language because we have unconsciously analyzed both the sentences syntactically as "SUBJECT VERB OBJECT" and after that syntax analysis, realized that "grass", being an inanimate thing, cannot be a SUBJECT for the VERB "eats". The operation we do is *semantic checking*. Thus, we note that in order to completely analyze a sentence, we had to do first syntax analysis and then semantic checking (and analysis) *controlled* by or *based on* that syntax analysis. This is what is meant by syntax-driven semantics. As semantic analysis is a step in the language translation, we do syntax-driven translation.

Further note that "grass" would be an allowed SUBJECT in sentence like "grass is green", i.e. whether "grass" is allowed as a SUBJECT in a sentence depends upon which VERB we have in that sentence. Thus, we have to consider relationships between the *types* of syntactical elements to do the semantic analysis.

Thus, we realize that

- Lexically and syntactically correct programs may still contain other types of errors.
- Lexical and syntax analyses are not powerful enough to ensure correct usage of variables, functions, statement constructs, etc. For example,

```
int a;
a = 1.345;
```

This error cannot be caught by the Scanner or the Parser.

The goal of semantic analysis is to ensure that the program satisfies a set of rules regarding the usage of various programming elements like variables, constants, functions and control constructs, etc. A typical partial set of such rules is given below:

- Variables must be defined before being used.
- A variable should not be defined multiple times.
- The same identifier cannot be used to denote different types of syntactic objects.
- In an `IF` statement, the condition expression must have Boolean type.
- In an assignment statement, the LHS must be a variable and the RHS expression must have the same type as LHS (or, in some languages like C, the RHS can be *coerced* into the type of LHS).

The semantic analysis is of two types – static (i.e. when the program is being compiled) and dynamic (i.e. when the program is running). The static semantic analysis is most useful for usual languages. It can be used to the following ways:

- Ensure validity of input programs. If we do not make these tests at compile time, the program may not work correctly.
  - Check that all variables and functions that are used in a program have been defined.
  - Check that the correct number and type of arguments are passed to functions and procedures.
  - Check that all variables are initialized before they are used.
- Clarify potential back-end ambiguities. If we do not do such an analysis, then it might be difficult or impossible to compile the programs.
  - Distinguish between different uses of the same variable name, for example, global and local.
  - Distinguish between different uses of the same symbols, for example, arithmetic operations on integers, floats, doubles, etc.
- Detect possibility for back-end optimizations. If we do not do this type of analysis, then we might not get the expected performance for the programs.
  - Identify redundant computations.
  - Identify repeated computations.
  - Allow run-time type checks to be omitted.
  - Efficiently use the CPU's registers and other computational resources.

Under the umbrella of semantic analysis, we have semantic rules regarding *types, scope* (i.e. regions of validity) and other semantic rules. We shall discuss the type checking in Chapter 6.

Most modern High Level languages contain considerable context-sensitive constructs. One way to handle this checking is to use a context-sensitive grammar to define our language and build a context-sensitive parser, but this is difficult and inefficient as of present-day state-of-art. That is why we use a separate semantic analysis phase to do the job. This phase gets activated generally after each

statement is parsed, but it can be interspersed within partial statement also, for example, during an expression parsing.

Another matter to be remembered is that the Symbol Table entries, which get initiated by the Scanner, get updated with new and additional information as the parsing and semantic analysis progress. The Symbol Table also gets referred to during these phases.

Semantics are of two types:

- **Static semantics:** The semantics of each program element (variable, function, label, etc.) remains fixed during the program execution.
- **Dynamic semantics:** The semantics of some of the program elements may change during the program execution.

Languages with static semantics allow considerable checking and error detection during compilation and are generally preferred. Java is a good example of such languages. Languages which are specified using dynamic semantics are more flexible to use, but require more careful programming, as their compilers would not have chance to catch many semantic errors.

There are several ways in which the semantics can be specified. Let us consider as example a "Let-expression" in some language. Its syntax is

$$E ::= let\ x = E_1\ in\ E_2$$

The following are various methods of specifying the semantics of such a statement:

Informal semantics: The occurrence of x in the expression E_2 denotes the value E_1. The whole expression E takes the value of E_2. This method is useful for human understanding and human checking of the implemented semantics.

Attribute grammar: Attribute *val* of E denotes a value. Attribute *env* binds variables to values. Thus, operation `bind(x,v,env)` creates a new environment with x in it bounds to v, and the bindings of the remaining variables are as in *env*. The specification for the "Let" is

$$E.val \Leftarrow E_2.val$$
$$E_1.env \Leftarrow E.env$$
$$E_2.env \Leftarrow bind(x, E_1.val, E_1.env)$$

Operational semantics: An interpreter `eval` takes two parameters – an expression and an environment. "Let" is specified as

$$eval(E, env) \Leftarrow (E_2, bind(x, eval(E_1, env), env))$$

This method is used for dynamic semantics.

Denotational semantics: The meaning of E, written as $[\![E]\!]$, is a function from environments to values. Thus, $[\![E]\!]\ env$ is application of $[\![E]\!]$ to environment *env*, is a value. For our example "Let",

$$[\![\text{let } x = E_1 \text{ in } E_2]\!]\ env = [\![E_2]\!]\text{bind}(\text{x}, [\![E_1]\!]\ env, env)$$

Each of the above methods has corresponding implementations. We shall consider only one of them – the attribute grammar method – in Section 5.4.

We shall first see how the semantics, represented by the generated code, is implemented within the Parser as it goes through the parsing steps. The purpose of discussing implementation before theory of attribute grammars is to provide a background in which the theory can be understood

better. We shall discuss *Implicit Stacking in an RDP, Synchronized Semantic Stacks* and *Action Symbols*.

5.1 Implicit Stacking in RDP

If the language can be parsed by an RDP, then a simple method is available to implement many of the semantic analysis operations. The method illustrated here is further used extensively in Chapter 8 on Intermediate Code, for generating the Intermediate representation of the program. Several further examples of implicit stacking in RDP are given.

We shall use the following sub-grammar of an IF-THEN-ELSE construct:

```
if-stmt   : IF logic-expr THEN stmt-list FI
          | IF logic-expr THEN stmt-list1 ELSE stmt-list2 FI
          ;
stmt      : basic-stmt
          | if-stmt
          ;
stmt-list : stmt
          | stmt-list stmt
          ;
```

This grammar may have to be modified depending upon the type of the parser used. The above grammar implies the following source construct and corresponding implied actions:

Source code:

```
IF logic-expr THEN stmt-list FI (next:)
```

Implied action:

{compute logical expr} {BranchFalse next:} {execute stmts} {next: ...

Source code:

```
IF logic-expr THEN stmt-list1 ELSE stmt-list2 FI (next:)
```

Implied action:

{compute logical expr} {BranchFalse label1:} {execute stmts-1} {Branch next:}

{label1: stmts-2} {next: ...

The execution of the logic expression puts a value True or False on the operand stack, where it can be tested by a Branch-if-False instruction. Note that the grammar implies possibility of nested IF statements at any reasonable depth of nesting. This will require that the labels like `label1` or `next` be "stacked", so that they are accessed only at the correct point in the parse tree. We shall see below how this is achieved.

IF c1 BF next1 s1 IF c2 BF next2 s2 FI next2: s3 FI next1:

As shown above, the labels are to be stacked because they are generated in a forward order, but needed in LIFO order.

Some of addresses (position of the labelled code) corresponding to these labels are not known when they are first encountered. For example, when generating the {Branch next:} action, the position value corresponding to the label `next:` is not known. Various "tricks" are used to deal with this situation.

The code in an RDP concerned with parsing an IF statement looks something like this:

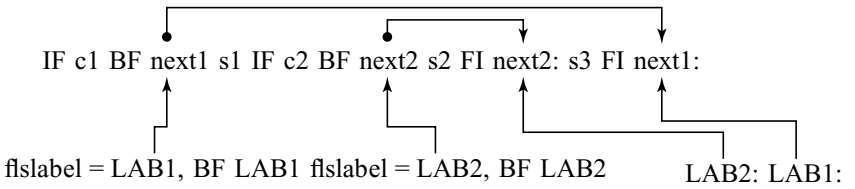
```
statement(){
  if(nexttoken() == IF) if_statement();
  else basic_statement();
}
statement_list(){
  while(nexttoken() != EOF){
    statement();
  }
}
if_statement(){
  pIF(); pE(); pTHEN(); statement_list();
  if(nexttoken() == FI) pFI();
  else{
    pELSE(); statement_list(); pFI();
  }
}
```

In order to provide the Intermediate code generation as a result of semantic analysis, we have to add code generation statements and local variable to the code for if_statement() as shown below.

```
if_statement(){
  char brlabel[10], flslabel[10];
  pIF(); pE();
  strcpy(flslabel, genlabel());     // added
  emit("BF", flslabel);             // added
  pTHEN(); statement_list();
  if(nexttoken() == FI){
    pFI();
    emit(flslabel, ":");
  } else{
    pELSE();
    strcpy(brlabel, genlabel());    // added
    emit("BR", brlabel);            // added
    emit(flslabel, ":");            // added
    statement_list(); pFI();
    emit(brlabel, ":");             // added
  }
}
```

In the above code, `genlabel()` creates unique labels every time it is called and `emit()` puts its argument strings in the output stream.

This function will be called two times recursively while parsing the nested IF statement given above, and the generated labels and Branch operations will be as shown below.



We can see that proper nesting of the IF statements will take place.

Note that the stacking of the generated code (here in this example the labels) was done implicitly in the local variables of the recursive function dealing with the language construct and no separate stack was used for the purpose. Also, parameter passing is used between various semantic routines to pass information about the semantics of the code being handled. The code is generated *after* a particular syntactic construct is detected.

5.2 Synchronized Semantic Stacks

For context-free grammars, being more general than the restricted grammar with which can be handled by an RDP, we have to use a separate stack to store the semantic information generated during the semantic analysis process. RDP used the return stack of the implementation platform to store this information in local variables created automatically on the return stack. Generally, LL(k), LR(1) or LALR(1) parsers do not have this one-to-one relationship between the Activation Record on the return stack and detection of a particular syntactic construct.

The separate stacks, more than one of which may be used by a parser, must be kept in synchronization with the detection of a particular syntactic construct. Such stacks are called synchronized Semantic stacks (Fig. 5.1).

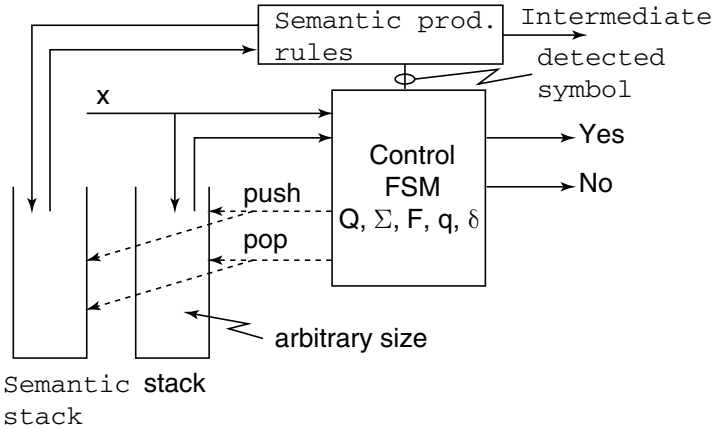


Fig. 5.1 Synchronized Semantic stack used for semantic analysis. Note that the Semantic stack does not push or pop every time the return stack does

Remember that ultimate goal of the semantic analysis is to generate the Intermediate representation (Chapter 8). Thus, the actual details of design and implementation of a semantic analyzer will be governed by the nature of the Intermediate representation.

To illustrate the idea, we implemented a Semantic stack with the RDP for the example grammar, as a part of a more extensive grammar given in Chapter 8. The C code is given in file `tb_rdp_if_sstk2.c`, some essential code segments from which are given here.

In anticipation of later development, we use the following definition of a `Node`, defined in file `tb_tree_l.h` as elements of the Semantic stack.

```
typedef union{
  struct node_struct * N;
  char  *S;
  long   I;
  float  F;
  void  *P;
}util;
typedef struct node_struct{
struct node_struct *link[2]; // left = 0, right = 1
util u,v,w,x,y,z;
}Node;
```

Note that we have defined the `util` union to give us a great flexibility in associating almost any kind of data in the Semantic stack. Though the node structure is optimized for a binary tree, it is flexible enough to construct any general tree. This can be done as follows:

```
link[0] = first child
link[1] = next sibling

                        [n]
                       /   \
             [p]=n->link[0]  p->link[1]
                     /  \
                    /    \

                   /      \
            [q]=p->link[0] [r]=q->link[1]
                           [p]->link[0]->link[1]  ... etc.
```

The original RDP for the example grammar is extended to handle the Semantic stack, by adding stack management functions and statements to obtain an RPN output. The Semantic stack is defined below:

```
#define SSTK 100
typedef struct semstk{
  Node *s[SSTK];
  int ssp;         // next empty cell
} sstk;

sstk semstack;
```

The stack push and pop functions are

```
int sstk_push(sstk *s, Node *p){
  if(s -> ssp > (SSTK - 1)){
```

```
      fprintf(stderr, "SemStack Full\n");
      return SSTK;
  } else {
      s -> s[s -> ssp] = p;
      (s -> ssp)++;
      return s -> ssp;
  }
}
Node *sstk_pop(sstk *s){
  if(s -> ssp <= 0){
      fprintf(stderr, "SemStack Empty\n");
      return NULL;
  } else {
      (s -> ssp)--;
      return s -> s[s -> ssp];
  }
}
```

Generally, we shall put character strings in the Semantic stack and we have to form a node out of them. The function to do that is

```
Node *tree_root(char *label){
  Node *np;
  np = (Node *) malloc (sizeof(Node));
  if(np == NULL)
    exit(-1);
  np -> u.N = NULL;
  np -> v.N = NULL;
  np -> w.S = (char *) malloc(strlen (label) + 2);
  strcpy(np -> w.S, label);
  strcat(np -> w.S," "); // to space when we print out
  return np;
}
```

We also require, for debugging our code, a stack dump function:

```
void sstk_displ(sstk *s){
  int n, i;
  n = s -> ssp;
  printf("sem.stack size = %d\n", n);
  for(i = 0; i < n; i++){
      printf("%s\n", s -> s[i]-> w.S);
  }
}
```

Some of the Terminal handling functions do not push or pop anything on the Semantic stack. They are: `pSEMI()`, `pIF()`, `pFI()`, `pELSE()`, `pTHEN()`, `plp()`, `prp()`. They simply check that the symbol read is as required by the syntax. We shall call them *delimiter functions*, as they possibly indicate some action within the RDP, but do not deal with the Semantic stack.

For example,

```
pIF(){
  printf("Entry: pIF()\n");
  if(symbol() != IF){
    fprintf(stderr, "Expecting I");
    exit(-1);
  }
}
```

On the other hand, there are Terminals handling functions which push a string on the Semantic stack. They are: `pASSGN()`, `pa()`, `pplus()`, `ppast()`. For example,

```
pASSGN(){
  printf("Entry: pASSGN()\n");
  if(symbol() != ASSGN){
     fprintf(stderr, "Expecting =");
     exit(-1);
  }
  sstk_push(&semstack,tree_root("="));
}
pa(){
  printf("Entry: pa()\n");
  int c;
  char str[4];
  if((c = symbol()) < 'a' || c > 'z' ){
     fprintf(stderr, "Expecting a-z");
     exit(-1);
  }
  sprintf(str,"%c",c);
  sstk_push(&semstack,tree_root(str));
}
```

Note that the function `pa()` allows us to use any small alphabetic character as a variable or number name.

Here are the rules by which the Semantic stack is operated:

Terminal functions (non-delimiters): Push the semantic value of the Terminal token, which generally is itself, on the Semantic stack. We have already seen examples of this rule above.

Non-Terminal functions: Just after calling a non-delimiter Terminal function pop the stack and if required by the output Intermediate language, hold the value in a temporary variable, to be issued to the output later. For example, code for `pE()` is given below:

```
pE(){
  Node *np;
  printf("Entry: pE()\n");
  pT();
  while(nexttoken() == '+'){
    pplus();
    np = sstk_pop(&semstack); // hold
```

```
    pT();
    strcat(out, np -> w.S); // to get RPN
  }
}
```

The code for `pT()` is similar.

Non-Terminal functions: Some of them will have to manipulate the order of tokens extracted from the Semantic stack, in order to get a proper RPN. For example,

```
basic_statement(){
  Node *np1, *np2;
  printf("Entry: basic-stsmt\n");
  pa();
  np1 = sstk_pop(&semstack);        // LHS of assignment
  pASSGN();
  np2 = sstk_pop(&semstack);        // '='
  pE();
  strcat(out, np1 -> w.S); strcat(out, np2 -> w.S);
}
```

Non-Terminal functions: It require additional semantic information to be inserted in the Intermediate representation output, for example, IF statement:

```
if_statement(){
  char lab1[10], lab2[10];
  printf("Entry: IF-stsmt\n");
  pIF();
  pE();                                     // expr
  strcpy(lab1,genlabel());
  strcat(out,lab1); strcat(out, "BF ");
  pTHEN();
  statement_list();                         // stmt1
  if(nexttoken() == FI){
     pFI();
     strcat(out,lab1); strcat(out,": ");
  } else {
     pELSE();
     strcpy(lab2,genlabel());
     strcat(out,lab2); strcat(out, "BR ");
     strcat(out,lab1); strcat(out,": ");
     statement_list();                      //stmt2
     pFI();
     strcat(out,lab2); strcat(out,": ");
  }
}
```

Utility functions: For example, a function to generate unique labels for control constructs like IF, WHILE, GOTO, etc.:

```
int nextlabel = 1;
char *genlabel(){
  char *s;
  s = malloc(8);
  sprintf(s, "LAB%d ", nextlabel++);
  return s;
}
```

In our example program `tb_rdp_if_sstk2.c`, the `main()` function, besides doing routine setup, reads the input string and prints out the final Intermediate representation in RPN.

We gave a string `a = s * (d + f)` as input and received RPN `s d f + * a =` as output. The following trace of the program shows the working of the synchronized Semantic stack.

```
                     Return-Stack    Semantic-Stack    In calling func.
Entry: stsmt-list
Entry: stsmt         push
Entry: basic-stsmt   push
Entry: pa()          push,pop        push(a),pop       hold1(a)
Entry: pASSGN()      push,pop        push(=),pop       hold2(=)
Entry: pE()          push
Entry: pT()          push
Entry: pF()          push
Entry: pa()          push,pop,pop    push(s),pop       out(s)
Entry: ppast()       push,pop        push(*),pop       hold3(*)
Entry: pF()          push
Entry: plp()         push,pop
Entry: pE()          push
Entry: pT()          push
Entry: pF()          push
Entry: pa()          push,3 pop's    push(d),pop       out(d)
Entry: pplus()       push,pop        push(+),pop       hold4(+)
Entry: pT()          push
Entry: pF()          push
Entry: pa()          push,3 pop's    push(f),pop       out(f)
Entry: prp()         push,4 pop's                      hold4 to out(+)
                            pop                        hold3 to out(*)
                          2 pop's                      hold1 to out(a)
                                                       hold2 to out(=)
DONE
s d f + * a =
```

If we wanted different types of Intermediate representation, we would have manipulated the Semantic stack differently (Chapter 8).

5.2.1 Semantic Stack in `yacc`

We have already introduced a well-known parser generator `yacc` in Chapter 4, Section 4.4, and introduced there the basic semantic facilities in `yacc`.

Internally, yacc maintains two stacks in memory – a *parse* stack and a *value* (or attribute) stack. The parse stack contains Terminals and non-terminals and represents the current parsing state.

The value stack is an array of YYSTYPE elements, and it associates a value with each element in the parse stack. For example, when lex returns an INTEGER token, yacc shifts (pushes) this token to the parse stack. At the same time, the corresponding `yylval` is shifted to the value stack. The parse and value stacks are always synchronized, so finding a value related to a token on the stack is easily accomplished. Here is a portion of the `yacc` input specification for a calculator:

```
expr:
       INTEGER          { $$ = $1; }
       | expr '+' expr { $$ = $1 + $3; }
       | expr '-' expr { $$ = $1 - $3; }
       ;
```

The grammar for `expr` is augmented by adding additional terms, called *Action* terms or symbols (Section 5.3). These action terms are enclosed in {}. Do not confuse these curly brackets with those used in C language, though the action term does generate a corresponding C code.

Here within the action terms, the `yacc` symbol `$$` denotes the semantic value associated with the LHS of a grammar rule. Symbols `$1` and `$3`, respectively, represent the values associated with the 1st and 3rd terms in a grammar rule, i.e. the two `expr` terms.

In the example, the augmented grammar says that if an `INTEGER` is detected, the value to be assigned to LHS (`$$`) is that given by the Scanner as `yylval ($1)`. If `expr '+' expr` is parsed, the value to be assigned to LHS is to be obtained by adding the value of the first `expr` and the second `expr`. Similarly for `expr '-' expr`, actually what happens is this: when `yacc` applies the rule,

```
expr: expr '+' expr { $$ = $1 + $3; }
```

it replaces the RHS of the production in the parse stack with the LHS of the same production. In this case, it pops `expr '+' expr` and pushes `expr`. It has now reduced the stack by popping three terms off the stack and pushing back one term. The `yacc` symbol `$$` designates the top of the value stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack and pushes back a single sum. Thus, the parse and value stacks remain synchronized.

The value (or the *attribute* in theoretical literature) assigned to the LHS of a grammar rule by the method discussed above is called *Synthesized attribute*, as it is created out of the attributes of the RHS of the same rule, see Section 5.4.4 for more details.

In fact, `yacc` has some more facilities for accessing the semantic values, which we shall discuss at an appropriate place.

There is one aspect of the Semantic stack in `yacc` which we have not mentioned till now. The semantic value of a grammar symbol can be almost anything:

- an integer, a float, a memory address or pointer, a string, even a data structure
- how does `yacc` handle that?

The value (Semantic) stack is formed as an array of cells of type `YYSTYPE`, which is also the type of `yylval`. In `yacc`, the input file defines `YYSTYPE` by a union construct similar to, for example,

```
%union {
  int i;       /* integer value      */
  float f;     /* float value        */
```

```
  char s;     /* symbol table index */
  Node *np;   /* node pointer       */
};
```

This results in the following code being included in the `yacc` output header file `y.tab.h`:

```
typedef union {
     int i;        /* integer value      */
     float f;      /* float value        */
     char s;       /* symbol table index */
     Node *np;     /* node pointer       */
} YYSTYPE;
extern YYSTYPE yylval;
```

The type of semantic value for each type of term in the grammar is defined at the beginning of the `yacc` input file as:

```
%token  <i> INTEGER
%type   <f> expr
```

This binds `expr` to `f`, and `INTEGER` to `i` in the YYSTYPE union. With this binding, `yacc` automatically generates a correct code, selecting a proper union member from `YYSTYPE`.

5.3 Action Symbols

The action terms specify the semantic rules by which value of LHS of a grammar rule is computed in terms of the values on the RHS. The augmented grammar simultaneously specifies two aspects – syntax and semantics – bare grammar rules specify the syntax and the action symbols specify the semantic rules.

For example, for the partial grammar for `expr`, the action symbols for generating RPN as the Intermediate code can be shown as below:

```
E : T { + <pop> T <out> }
T : F { * <pop> F <out> }
F : a <pop> <out> |( E )
```

with a proviso that Terminals '+', '*' and '*a*' push themselves. Compare this against the example code given in Section 5.2. Note that the nature of interpretation and position of the action symbols depend upon the output we want.

5.3.1 Action Symbols in `yacc`

Symbol Values

Every symbol in a `yacc`-based parser has a value. The value gives additional information about a particular instance of a symbol. For example, if a symbol represents a number, the value would be the particular number. If it represents a literal text string, the value would normally be a pointer to a copy of the string. If it represents a variable in a program, the value would be a pointer to a symbol table entry for the variable. Some tokens do not have a useful value, e.g. a token representing a closed

parenthesis, it is a delimiter – it may trigger some action, but has no *value*. Non-terminal symbols can have any values you want, as decided by the semantics of the language and created by the action code in the parser. Usually, the action code builds a semantic tree corresponding to the input, so that later phases can process a whole statement or even a whole program at a time.

In real parsers, the values of different symbols use different data types, e.g. `int` and `double` for numeric symbols, `char *` for strings. We have seen previously that `yacc` handles multiple value types by creating a C `union typedef` called `YYSTYPE` to contain them.

Whenever the parser reduces by a rule, it executes user-specified C code associated with the rule, known as the rules *action*. The action appears in braces after the end of the rule, before the semicolon or a vertical bar. The action code can refer to the values of the RHS symbols in the rule as $1, $2, etc. and can set the value of the LHS by setting $$.

Embedded Actions in `yacc`

Sometimes we have to get some action be executed during the recognition of a grammar rule. Embedded actions in `yacc` provide this facility. Rules in `yacc` may contain embedded actions:

```
list: item1 { do_item1($1); } item2 { do_item2($3); } item3
```

Note that the actions take up a cell in the Semantic stack, so `do_item2` must use `$3` to reference `item2`. Actually, this grammar is transformed by `yacc` into the following:

```
list: item1 _rule01 item2 _rule02 item3
_rule01: { do_item1($0); }
_rule02: { do_item2($0); }
```

5.4 Attribute Grammars

The processing required to build up the meaning of a higher syntactic construct from the meaning of a sequence of lower constructs is formalized by the use of *attribute grammars*.

We augment a grammar by associating attributes with each grammar symbol that describes its properties. An attribute has a name and an associated value, which may be a string, a number, a type, a memory location, an assigned register – whatever information we need to associate with that symbol in order to specify its meaning. For example, variables may have an attribute *type*, which keeps track of the declared type of a variable, useful later in type-checking or an integer constant may have an attribute numerical *value*, which we will later need to generate the target code. With each production in a grammar, we give semantic rules or actions, which describe how to compute the attribute values associated with each grammar symbol in a production. The attribute value for a parse node may depend on information from its children nodes below (synthetic attribute) or its siblings and parent node above (inherited attribute).

For example, consider the following production, augmented with a set of actions that use the *value* attribute for a digit node to store the appropriate numeric value. We use the syntax `X.a` to refer to the attribute `a` associated with a symbol `X`.

```
digit        : 0   {digit.value = 0}
             | 1   {digit.value = 1}
             | 2   {digit.value = 2}
             ...
```

```
            | 9   {digit.value = 9}
            ;
```

This production says: syntax-wise a "digit" is either 0, 1, …, or 9 and semantics-wise value of the digit is as indicated by the *action* term specified in curly brackets. Note well that in clauses like `1 {digit.value = 1}` the first '1' denotes simply a symbol for unity but the second '1' denotes value 1 as we normally understand in arithmetic. Similar comments apply to the other clauses.

Attributes may be passed up a parse tree to be used by other productions, for example,

```
int1        : digit        {int1.value = digit.value}
            | int2 digit   {int1.value = int2.value * 10 + digit.value}
```

We have used names `int1`, `int2` in this example to clarify which attribute we are referring to, so `int1` and `int2` are different instances of the same non-terminal symbol `int`. Note that in the above production we have defined the value of an integer expressed in decimal notation.

5.4.1 Syntax-directed Techniques

Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. This can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars.

5.4.2 Definition of Attribute Grammar

An attribute grammar is a BNF grammar with certain additions:

- For any grammar symbol X: a set A(X) of attribute values.
- Each production in the grammar has a set of semantic rules that define or compute certain attributes of the non-terminals in the production.
- Each production in the grammar has a set, possibly empty, of predicates to check for attribute consistency.
- A sentence derivation based on BNF gives a parse tree.
- A sentence derivation based on an attribute grammar gives a fully *attributed parse tree* or an *annotated parse tree.*

There are two types of attributes we might encounter: *synthesized* or *inherited.* Synthesized attributes are those attributes that are passed up a parse tree, i.e. the LHS attribute is computed from the RHS attributes. The lexical analyzer usually supplies the attributes of Terminals and the synthesized ones are built up for the non-terminals and passed up the tree.

Inherited attributes are those that are passed down a parse tree, i.e. the RHS attributes are derived from the LHS attributes, or other RHS attributes. These attributes are used for passing information about the context to nodes further down the tree.

Thus, synthesized attributes are concerned with passing meaning derived from the structure and inherited attributes are concerned with meaning derived from the context.

Syntax-directed Definitions

Syntax-directed definitions mean attaching a semantic rule to each production rule in the parse grammar. Syntax-directed definitions use CFG to specify the syntactic structure of the construct. It

associates a set of attributes with each grammar symbol, and with each production, it associates a set of semantic rules for computing the values of the attributes of the grammar symbols appearing in that production. Therefore, the grammar and the set of semantic rules constitute syntax-directed definitions.

5.4.3 Dependency Graphs

Given an attribute grammar, each rule has an associated *dependency graph.* This graph has a node for each attribute $A_j(X_i)$ of each symbol X_i in the rule. For each attribute rule of the form

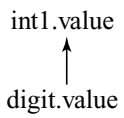
$$A_j(X_i) = f(A_p(X_0), \ldots, A_q(X_k), \ldots))$$

there is one edge from each node $A_q(X_k)$ on the RHS, to the node $A_j(X_i)$, which denotes the dependency of $A_j(X_i)$ on $A_q(X_k)$.

For example, consider the decimal integer grammar given in Section 5.4. As there is only one attribute – the `value` – to be considered, there is only one node per dependency graph. In the grammar, which we reproduce here:

```
int1     : digit       {int1.value = digit.value}
         | int2 digit  {int1.value = int2.value * 10 + digit.value}
```

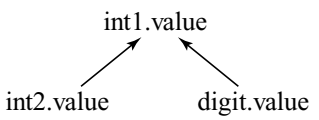
For the first alternative rule, the dependency graph is



The second alternative production has associated attribute equation

```
int1.value = int2.value * 10 + digit.value
```

The dependency graph for this rule is



5.4.4 Definitions: Inherited and Synthesized Attributes

Let $X_0 \rightarrow X_1 \ldots X_n$ be a production.

- If the computing rule of X_0's attribute is of the form

  $A(X_0) = f(A(X_1), \ldots, A(X_n))$

  then it is a synthesized attribute.
- If the computing rule of X_j's attribute is of the form

  $A(X_j) = f(A(X_0), \ldots, A(X_i), \ldots, A(X_{j-1}))$, for $i \leq j \leq n$

  then it is an inherited attribute. Or $A(X_j) = f(A(X_0), \ldots, A(X_i), \ldots, A(X_{j-1}), A(X_j))$.
  Inherited or synthesized attributes associated with X_j itself can be present but without cycles in the dependency graphs.
- Terminals have *intrinsic attributes*, i.e. the *Lexical values* supplied by the Lexical analyzer.

As an example, inherited attributes for an expression grammar are shown in Fig. 5.2.

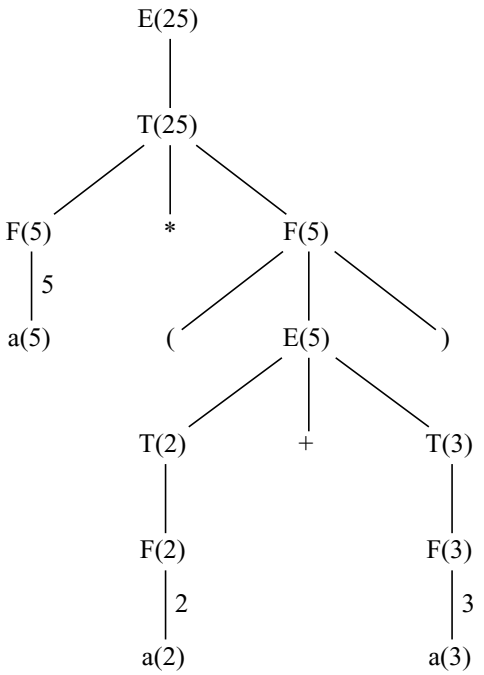


Fig. 5.2 Example of inherited attributes

Here the attributes are the values of the terms of an expression. The attribute inheritance starts Bottom-up, i.e. from RHS to LHS in the grammar. The corresponding attribute grammar is given below:

```
E0 : E1 + T    E0.value = E1.value + T.value
E  : T         E.value = T.value
T0 : T1 * F    T0.value = T1.value + F.value
T  : F         T.value = F.value
F  : ( E )     F.value = E.value
F  : a         F.value = a.value
```

On the other hand, consider the following grammar for C language style variable declaration:

```
D : T L
L : L , I
T : int | float
I : x | y | z
```

Now consider a declaration `int x, y, z`. The synthesized and inherited attribute derivation is shown in Fig. 5.3.

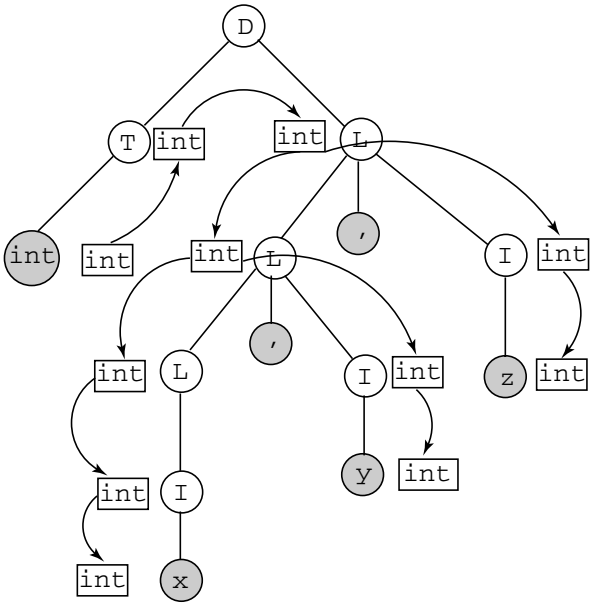


Fig. 5.3 Example of inherited attributes. One up-going arrow denotes a synthesized attribute, and all remaining arrows indicate inherited attributes

The corresponding attribute grammar is given below:

```
D  : T L       L.in     = T.type        (inherited)
T  : int       T.type   = int.int       (synthesized)
T  : float     T.type   = float.float   (synthesized)
L0 : L1 , I    L1       = L0.in         (inherited)
L  : I         I.in     = L.in          (inherited)
I  : id        id.type  = I.in          (inherited)
```

5.4.5 S-Type Definitions and Grammars

For synthesized attributes, we perform bottom-up tree traversal for attribute evaluation. A syntax-directed definition (SDD) is S-attributed if every attribute is synthesized in that traversal.

5.4.6 L-Type Definitions and Grammars

For SDDs with both inherited and synthesized attributes, we have to look at the *dependency graphs.*

Dependency Graphs

They determine how attributes can be evaluated in a parse tree. For each symbol *X,* the dependency graph has a node for each attribute associated with *X.* An edge from node *A* to node *B* means that the attribute of *A* is needed to compute the attribute of *B*.

How to differentiate the synthesized attributes from the inherited attributes?

L-attributed SDD: An SDD is L-attributed if in all of its dependency graphs the edges only go from left to right but not from right to left, and there is no circular dependency. This will guarantee that there is an evaluation order.

A typical case of circular dependency is given below:

```
Production    Semantic rules
A -> B        A.synth = B.inher
              B.inher = A.syth + 1
```

5.4.7 Synthesized and Inherited Attributes in yacc

The yacc example we considered in Section 5.2.1 used synthesized attributes. At any point in a syntax tree, we can determine the attributes of a node based on the attributes of its children. Consider the rule

```
expr: expr '+' expr { $$ = $1 + $3; }
```

Since we are parsing bottom-up, the attributes of both the operands are available, and yacc can determine the value associated with the left-hand side.

An inherited attribute of a node depends on the value of a parent or sibling node. For example, the following grammar defines a C variable declaration:

```
decl: type varlist ;
type: INT | FLOAT ;
varlist: VAR { setType($1, $0); }
       | varlist ',' VAR { setType($3, $0); }
```

Here is a sample parse for such a declaration:

```
. INT VAR
INT . VAR
type . VAR
type VAR .
type varlist .
decl .
```

The dots denote the progress of the parsing pointer. When yacc reduces VAR to varlist, it should annotate the symbol table with the type of the variable. However, the type is buried in the stack. This problem is resolved by indexing back into the stack. Recall that $1 designates the first term on the RHS. We can index backwards, using $0, $-1 and so on. In this case, $0 will take yacc to type. If you need to specify a token type, the syntax is $<tokentype>0, angle brackets included. In this particular example, care must be taken to ensure that type always precedes varlist.

5.4.8 More on Inherited Attributes in yacc

Yacc symbol values can act as inherited or synthesized attributes. Entities which are referred to as attribute in theoretical literature, are termed values in yacc. Attributes get inserted as token values during the parsing process, i.e. as the leaves of the parse tree. Conceptually, the attribute information moves up the parse tree each time a rule is reduced and the rules' action synthesizes the value of its resulting symbol ($$) from the values of the symbols on the right-hand side of the rule.

Sometimes you need to pass information the other way, i.e. from the root of the parse tree towards the leaves. For example,

```
decl:   class type varlist ;
class : GLOBAL { $$ = 1; }
```

```
      | LOCAL { $$ = 2; }
      ;
type:   FLOAT { $$ = 10; }
      | INT   { $$ = 20; }
      ;
varlist: NAME         { insert($0, $-1, $1); }
      | varlist NAME  { insert(S0, $-1, $2); }
      ;
```

It would be useful to have the `class` and `type` available in the actions for `varlist`, both for type checking and to enter into the Symbol Table. In `yacc`, it is possible to access the symbols on its internal stack to the left of the current token, via $0, $-1, etc. In the example above, $0 in the call to `insert()` refers to the value of the symbol `type` which is stacked just before the symbo1/s for the `varlist` production and will have the value 1 or 2, depending on whether the `type` was `FLOAT` or `INT`. The second argument $-1 refers to the symbol `class` which will have the value 10 or 20 depending upon whether the class was `GLOBAL` or `LOCAL`.

Although inherited attributes as used in `yacc` can be very useful, they can also be a source of hard-to-find bugs. An action that uses inherited attributes has to take into account every place in the grammar where its rule is used. In this example, if you changed the grammar to use the symbol `varlist` somewhere else, you would have to make sure that in the new place where the symbol `varlist` occurs appropriate symbols precede it so that $0 and $-1 will get the right values.

Inherited attributes are useful especially for complex constructs like C language variable declarations. However, in many cases it is safer and nearly as easy to use synthesized attributes. In the example above, the `varlist` rules could create a linked list of references to the names to be declared and return a pointer to that list as its value. The action for declaration could take the `class`, `type` and `varlist` values and at that point assign the class and type to the names in the `varlist` and/or in the Symbol Table.

When you use the value of an inherited attribute, the usual value declaration, e.g. `%type`, does not work. Since the symbol corresponding to the value does not appear in the rule, `yacc` cannot decide what the correct type is. You have to supply type names in the action code using an explicit type. In the example above, if the types of `class` and `type` were `cval` and `tval`, respectively, the production for `varlist` would actually look like:

```
namelist: NAME { insert($<tval>0, $<cval>-1, $1); }
        | varlist NAME { insert($<tval>0, $<cval>-1, $2); }
        ;
```

5.5 Symbol Table Handling

We already had a preliminary discussion about Symbol Tables in Chapter 3, Section 3.2. A Symbol Table exists throughout the compilation steps. Major operations required for a symbol table are:

- insertion,
- search,
- modify contents of a node,
- deletions are purely logical (depending on scope and visibility) and not physical,
- keywords are often stored in the symbol table before the compilation process begins, so that Scanner and Parser are able to trap them.

The Symbol Table is accessed at every stage of the compilation process:

Scanning: Insertion of new identifiers.

Parsing: Access to ensure that an operand exists.

Semantic analysis:

- Determination of types of identifiers from declarations,
- Type checking to ensure that operands are used in type-valid contexts,
- Checking scope, visibility violations.

Intermediate Representation generation: Memory allocation and relative address calculation for jumps etc.

Optimization: All memory accesses are through the Symbol Table.

Target code generation: Translation of relative addresses to absolute addresses in terms of word length, word boundary, etc.

Thus, the Symbol Table is a store house of context-sensitive and run-time information about every identifier in the source program. That is why sometimes it is called the *Environment.*

All accesses relating to an identifier require to first find the attributes of the identifier from the Symbol Table. In real-world compilers, it is usually organized as a hash table, which provides fast access. Compiler-generated temporary memory locations may also be stored in the symbol table.

The following attributes are stored in a Symbol Table for each identifier:

- type,
- size,
- scope/visibility information,
- base address,
- a pointer to the identifier name string.

5.5.1 Symbol Table in `miniC`

After addition of semantic actions for type setting in the `yacc` grammar of the `miniC` language as shown below:

```
decl:    dtype ':' dlist ;
dtype:   IVAR | VAR | SVAR ;
dlist:   VAR                    { type($1) = type($<sym>0); }
       | dlist ',' VAR          { type($3) = type($<sym>0); }
       ;
```

For the following declarations,

```
int    : a
float  : f
string : s
```

we obtained the following Symbol Table dump:

```
[DEG](261){42652ee1}
[E](261){402df854}
```

```
[PI](261){40490fdb}
[a](275){0}            <---------+
[cos](262){8048b1c}              |
[else](267){0}                   |
[f](261){0}            <---+     |
                           |     |
[float](261){0}        <---+     |
[for](265){0}                    |
[if](266){0}                     |
[int](275){0}          <---------+
[s](276){0}            <---+
[sin](262){8048c0c}        |
[string](276){0}       <---+
[while](264){0}
```

For declaration of a function,

```
func ack(){
   return 1
}
```

we obtained the following entry in the Symbol Table

```
[ack](268){8050d80}
```

due to the following type setting action in the grammar:

```
defn: FUNC procname { type($2) = FUNCTION;}
          '(' ')' stmt { }
    | PROC procname { type($2) = PROCEDURE;}
          '(' ')' stmt { }
    ;
```

5.6 Intermediate Representation Output for `miniC`

Though it is possible to include actions in the `yacc` grammar of a language to either directly execute a program written in that language (as is done in `hoc6`) or directly generate a target code, we have taken the option of generating Intermediate Representation (IR) (Chapter 8) in anticipation of discussion of various optimization methods.

Any reasonable IR can be obtained by including proper action terms in the `yacc` grammar. Here we illustrate a small part of the final grammar with action terms for generating two kinds of IR – reverse polish notation (RPN) and 4-tuple notation. The full grammar with IR generating actions is given in Chapters 8 and 12. Here, we show IR for arithmetic expressions `expr` and assignment statement `asgn`.

The following miniC program, `test1.miniC` was used as an example:

```
int : a,b,c,d
a = 3
b = 5
```

```
c = 7
d = a * (b + c)
end
```

The RPN output was:

```
AST: 3
AST: a =
AST: 5
AST: b =
AST: 7
AST: c =
AST: a
AST: b
AST: c
AST: +
AST: *
AST: d =
```

The 4-tuple (matrix) output was:

```
MAT1:  LD   3    T1   --
MAT2:  =    T1    a   --
MAT3:  LD   5    T2   --
MAT4:  =    T2    b   --
MAT5:  LD   7    T3   --
MAT6:  =    T3    c   --
MAT7:  LD   a    T4   --
MAT8:  LD   b    T5   --
MAT9:  LD   c    T6   --
MAT10: ADD  T5   T6   T7
MAT11: MUL  T4   T7   T8
MAT12: =    T8    d   --
```

The corresponding action terms in the `yacc` grammar are:

```
asgn: VAR '=' expr {printf("AST: %s =\n", name($1));
      printf("MAT%d: = T%d %s --\n",++mcount,tcount,name($1));tpush(tcount);}
    | IVAR '=' expr { printf("AST: %s =\n", name($1));
      printf("MAT%d: = T%d %s --\n",++mcount,tcount,name($1));tpush(tcount);}

expr: INT {$$ = $1; printf("AST: %s\n", name($1));
           printf("MAT%d: LD %s T%d --\n", ++mcount,name($1),
                                      ++tcount);tpush(tcount);}
    | NUMBER {$$ = $1; printf("AST: %s\n", name($1));
           printf("MAT%d: LD %s T%d --\n", ++mcount,name($1),
                                      ++tcount);tpush(tcount);}
```

```
| VAR {$$ = $1; printf("AST: %s\n", name($1));
      printf("MAT%d: LD %s T%d --\n", ++mcount,name($1),
                                  ++tcount);tpush(tcount);}
| IVAR {$$ = $1; printf("AST: %s\n", name($1);
      printf("MAT%d: LD %s T%d --\n", ++mcount,name($1),
                                  ++tcount);tpush(tcount);}
| expr '+' expr { printf("AST: +\n"); $$ = $3;
      printf("MAT%d: ADD T%d T%d T%d\n", ++mcount, tpop(), tpop(),
                                  ++tcount);tpush(tcount);}
| expr '-' expr { printf("AST: -\n"); $$ = $3;
      printf("MAT%d: SUB T%d T%d T%d\n", ++mcount, tpop(), tpop(),
                                  ++tcount) ;tpush(tcount);}
| expr '*' expr { printf("AST: *\n"); $$ = $3;
      printf("MAT%d: MUL T%d T%d T%d\n", ++mcount, tpop(), tpop(),
                                  ++tcount); tpush(tcount);}
| expr '/' expr { printf("AST: /\n"); $$ = $3;
      printf("MAT%d: DIV T%d T%d T%d\n", ++mcount, tpop(), tpop(),
                                  ++tcount); tpush(tcount);}
```

Note that we have inserted the action terms for both RPN and 4-tuples at the same places in the grammar, though the nature of these two IRs is quite different.

? EXERCISES

1. For a reasonably complete semantic check on the variables in a program, the following information may need to be associated with each variable `a`, by keeping it in its Symbol Table entry.
   - Does the program include a declaration for `a`?
   - Does the program include a definition for `a`?
   - What is the type of `a`?
   - Has `a` been properly initialized (before use)?
   - Does any function or statement ever use `a`?
   - What is the storage address of `a`?
   - Is `a` in the current *scope*?

   Considering the `Node` definition used by us, suggest ways in which this requirement be satisfied.
2. Consider the grammar G_b given in the exercises of Chapter 4 for "Boolean expression calculator". Express its attribute grammar in `yacc` format.
3. Prepare an attribute grammar in `yacc` format for the grammar G_p of the "Parentheses Language" given in the exercises of Chapter 4. The attribute we want to express is the *depth of parenthesis.*
4. Draw and annotate a tree for input $((a)+(b))$ given a grammar with the following rules:

```
E -> E + T     $$ = mknode("+",$1,$3)
E -> E - T     $$ = mknode("-",$1,$3)
E -> T         $$ = $1;
T -> ( E )     $$ = $2;
T -> id        $$ = mkleaf(id,$1)
T -> num       $$ = mkleaf(num,$1)
```

`mknode(op,left,right)` creates an operator node with the label op and two pointers: left and right.

5. Consider the following grammar:

```
S -> L . L
S -> L
L -> L B
L -> B
B -> 0
B -> 1
```

Let the synthesized attribute `val` gives the value of the binary number generated by *S* in the grammar. For example, on input 101.101, S.val = 5.625. Use synthesized attributes to determine S.val.

6. Using `yacc`, implement a syntax-directed translator that translates the sequences of postfix polish expressions into infix notation. For example, your translator should map 345+* into 3*(4+5)
7. Optimize your translator so it does not generate any redundant parentheses. For example, your translator should still map 345+* into 3*(4+5), but it should map 345*+ into 3+4*5.
8. You are required to develop a program similar to the `indent` utility in Unix-like systems. This utility re-formats a C source code according to a specified coding style, especially, it provides nice indentations. For example, given a source file:

```
#include <stdio.h>
int main(){int i; printf("Start\n"); if(i < 10) printf("OK\n");}
```

it converts it into one looking like:

```
#include <stdio.h>
int
main()
{
  int i;
  printf("Start\n");
  if(i < 10)
    printf("OK\n");
}
```

Your program will have to parse the input file for at least the control constructs in C. Write the `yacc` grammar with required action terms to develop this utility.

9. In this exercise you are required to write a syntax highlighter program, which prints a given C source code, with all the keywords in bold-face and conditions within IF-THEN and WHILE-DO in italics. Your program will have to do some minimal parsing. You can assume that inserting <bf> </bf> around a word will make it display/print in bold-face and, similarly, <it> </it> will make a string display in italics. Write the required `yacc` grammar with proper action terms to achieve this utility.
10. School children are taught a simple, graphics-based programming language called `LOGO`. Obtain details of this language, if possible install it on your computer and play with it and then write an interpreter for `LOGO`.

11. In this exercise you are required to prepare the SDT scheme for a column-major organization of an array. Indicate the attributes of your SDT scheme along with the auxiliary functions you are using. Define a grammar and its semantic actions for this translation. Show the annotated parse tree and generated code for the assignment $x = A[y, z]$ under the assumption that A is 10×20 array with starting index 0 and sizeof(baseType(A)) = sizeof(int) = 4-bytes.

WEB RESOURCES

http://en.wikipedia.org/wiki/Attribute_grammar
http://en.wikipedia.org/wiki/Syntax-directed_translation

FURTHER READING

1. Aho, Lam, Sethi and Ullman, "Compilers – Principles, Techniques and Tools" 2nd Edition, Pearson Education.
2. Bennett J.P., "Introduction to Compiling Techniques", Tata McGraw Hill.

GLOSSARY

Binding Associating values and attributes with some identifier.
Inheritance Attributes acquired via biological heredity from the parents.
Intrinsic Belonging to a thing by its very nature.
Semantic-stack Stores the attributes of parsed symbols as parsing proceeds.
Siblings A person's brothers or sisters, in graph theory, nodes having the same parent node.
Synchronized Operating in unison; arrange or represent events so that they co-occur; operate simultaneously.
Trigger A device that activates or releases or causes something to happen, an act that sets in motion some course of events.
Unconsciously Without awareness about what is happening in the environment.
Validate Give evidence for; declare or make legally correct.
Violations An act that disregards an agreement or a right.

6

Type Checking

What you will learn in this chapter

- What are data types?
- What do we mean by type checking?
- Which components of a program need to be type checked?
- How do we infer the type for a component?
- What is the formalism used for type checking?
- What is type conversion and coercion and how is it done?
- How is overloading of operators and functions type checked?

Key Words

type checking, type inference, formal semantics, type conversion and coercion, overloading

We view type checking as evaluation with "abstract values". Whereas dynamic semantics – evaluation, deals with dynamically computed values like 0, 1, etc., static semantics – type checking – deals with abstractions like `int`. A type system is sound if it correctly approximates the dynamic behavior and predicts its outcome. For example, if the static semantics predicts that a term has the type `int`, the dynamic evaluation of the term, if it terminates, will yield an integer.

As indicated in Chapter 5, type checking is one important static semantics check. Many of the programmer errors can be detected and faster progress to a correctly running program is achieved by type checking. We shall now discuss this aspect of semantic analysis in detail.

Some languages like Java and Ada are strongly typed and its compiler enforces strict type checking. On the other extreme, languages like BCPL, or to some extent, Perl, have no type system and any value can be used anywhere in the program if it is semantically allowed, i.e., a reasonable meaning can be found. The C language stands somewhere in the middle of these two extremes.

The most usual type checks applied to a program are:

Identifiers: Do we know the type of this identifier?

Expression: Are the operands of each operator in an expression of the correct type?

Assignment: Can we assign the type of value of an expression, for example the RHS, to a variable with some other type, say on the LHS?

Statements: Is an expression appearing within a statement, of the correct type? For example, the condition expression within a while-loop should have type *Boolean.*

Conversion, Coercion: Is it possible to convert or *coerce* the type of an expression to the required type at this point in the program construct? Note that if such conversion or coercion is indeed allowed, then the semantic analysis phase will have to generate extra code to execute that conversion at run-time.

The types of identifiers, program constructs, values and rules about how these types are manipulated form the *type system* of a language.

6.1 Data Types and Type Checking

The usual data types are *integer, float* (we purposefully avoid calling them *real,* as they really are not), *character, string, boolean, label, pointer.* Some languages have sub-types within some of these types, for example, in C and Java we have *short, long* and *long long* sub-types associated with the type *integer.* Similarly, type *double* is a sub-type associated with the type *float.* Usually, arrays of these basic types are also available and form their own type. In many languages, data structures are available as complex data type.

In C, we can have user defined types for arbitrary data structures and complex data types built-up recursively for defined types. For example, in C we can have a declaration `float(*pf[]())` denoting "pf: an array of functions returning pointer to float". In spite of this complexity of data types in C, it does not enforce types strictly. For example, a programmer can declare a function without specifying its argument types and the compiler will happily allow any call argument type. That is why C is called a *weakly* typed language. A good C language compiler issues *warnings,* but not *errors,* when doubtful constructs are encountered.

Remember that the Symbol Table that we have been using in Chapters 3, 4 and 5, did have a field `type` to record the type of an identifier or number. For your ready reference we have given the Node data structure below.

```
typedef union{
  struct node_struct*N;
  char*S;
  long I;
  float F;
  void *P;
}util;
typedef struct node_struct{
struct node_struct *link[2]; // left = 0, right = 1
util u,v,w,x,y,z;
}Node;
#define name(n)    ((n)->w.S)
#define type(n)    ((n)->v.I)
```

We have also seen in Chapter 5 how the type of a declaration was transmitted to the variables being declared.

```
decl:    dtype ':' dlist ;
dtype:   IVAR | VAR | SVAR ;
```

```
dlist:   VAR                    { type($1) = type($<sym>0); }
       | dlist ',' VAR          { type($3) = type($<sym>0); }
       ;
```

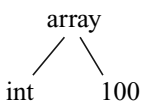
In languages like C which allow complex data types, we require more sophisticated methods for defining the type of an identifier or expression.

6.2 Type Expressions and Type Constructors

One obvious way of representing a complex data type is to show how it is arrived at by means of a tree structure. For example, consider the following declarations and definition.

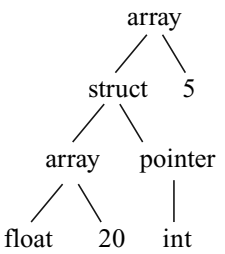
```
int a[100];
```

This can be represented by a tree as:




```
struct my {
  float f[20];
  int *pi;
} b[5];
```

This array of `struct` definition can be represented by a tree as:



Usually, a complex data type is constructed by the following methods or their combination:

Pointers: have a single component of type T, the type of construct is a pointer to T.

Arrays: have two components – type of array elements T (left child) and number of elements (right child).

Structures (Records): have one component (child) for each field in the structure, in the left-to-right order in which they appear in the structure.

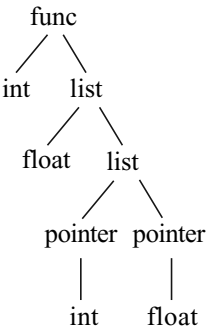
Functions: have two components (children) – the type of the returned value and the list of types of arguments.

Unions: similar to structures.

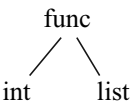
It will be desirable to construct these type trees as Binary trees, but here we face a problem – in general a structure can have more than two fields and a function could have more than one argument. We must have a way of representing more than two children in a Binary tree. This can be done as follows. Suppose we have a function with 3 arguments:

```
int func1(float a, int *b, float *c);
```

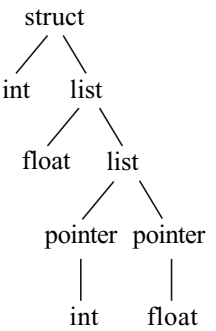
We can represent this by a tree such as:



A function without arguments will be represented by a tree such as:



A type tree for a structure with more than two fields be similarly constructed.
For example:



represents a `struct` with four fields:

```
struct four {
  int   i;
  float f;
  int   *pi;
  float *pf;
}
```

Though the Node structure we have suggested previously can easily be used for representing a non-Binary tree, we are still interested in Binary type tree, as we already have in our Scanner functions for Binary tree manipulations like insert node, look-up node, traverse, etc.

In the Symbol Table, we should be able to refer to a type tree instead of a simple type. First, we define additional types:

```
#define   INT          274
#define   NUMBER       258
#define   STRING       259
#define   VAR          261
#define   IVAR         275
#define   SVAR         276
#define   FUNCTION     268
#define   PROCEDURE    269
#define   TREE         1000
#define   POINTER      TREE+1
#define   ARRAY        TREE+2
#define   STRUCT       TREE+3
#define   UNION        TREE+4
#define   FUNCDEF      TREE+5
#define   LIST         TREE+10
```

Note that all the structuring types have assigned values greater than TREE = 1000. Type `TREE` is simply used in code to decide if we have a complex type definition.

We now redefine the interpretation of various fields in the Node structure as follows:

```
typedef union{
  struct node_struct*N;
  char*S;
  long I;
  float F;
  void *P;
}util;

typedef struct node_struct{
struct node_struct *link[2]; // left = link[0], right = link[1]
util u,v,w,x,y,z;
}Node;
#define L(n) ((n)->link[0])      \
#define R(n) ((n)->link[1])      |
#define left(n)    L(n)          |  Basic Symbol Table
#define right(n)   R(n)          |
#define name(n)    ((n)->w.S)    /

#define type(n)    ((n)->v.I)  --+    Types:
                               |    if > TREE than a constructor
#define ttree(n)   ((n)->z.N)  <-+    tree at this pointer
```

Other fields in the Node are used for other purposes and not of concern here.

6.3 Type Equivalence

While type checking, we will have to check *equivalence* of types. For simple types, it is a matter of merely comparing `type(np1)` with `type(np2)`, where `np1` and `np2` are pointers to nodes representing the two program components.

When we have complex data types, we need to ensure that the two type trees have exactly the same structure. We do this by a `int tree_comp(Node *np1, Node *np2)` function:

```
int tree_comp(Node *np1, Node *np2){
   if(type(np1) == type(np2)){
      if(type(np1) < TREE) return 1;
      retrun tree_comp_rec(ttree(np1), ttree(np2));
   }
   return 0;
}
int tree_comp_rec(Node *np1, Node *np2){
   return tree_comp(np1, np2);
}
```

6.4 Type Names, Declarations and Recursive Types

Many languages like C allow user defined named complex types. In C, this is specified via `typedef` statement. You must have seen and used several examples of this kind of named types.

How do we check equivalence if named types are used?

Two approaches are possible:

Named equivalence: Treat named types as simple types and just check that two program components have the same name.

Structural equivalence: Replace the named type by their definitions (which may be a definition tree) and recursively check those trees.

For example, let us define two user-defined data types:

```
typedef float * series;
typedef float * sequence;
```

These two data types `series sequence` are different for named equivalence viewpoint, but same from structural equivalence consideration. Which of these two types of equivalence should we use? The answer is related to another facility of type definitions in many languages, called *Recursive* types.

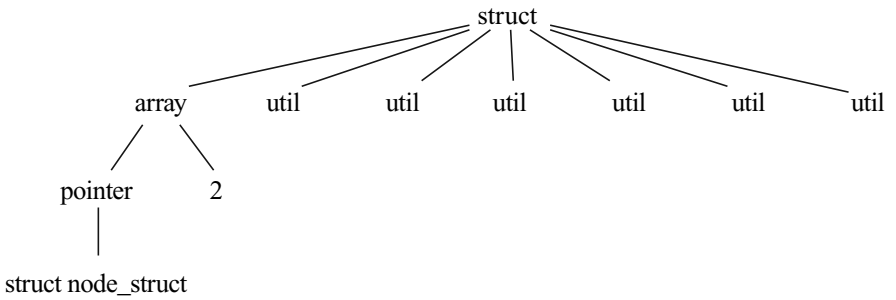
6.4.1 Recursive Types

Consider our definition of `Node`:

```
typedef struct node_struct{
  struct node_struct *link[2];
  util u,v,w,x,y,z;
} Node;
```

The definition contains a self-reference while defining the `link[2]` fields, as the `link` array is defined as array of pointers to this `struct` itself. Thus the definition of Node is called a *Recursive* type.

If we try to represent the definition of `Node` as a graph having back-edges from `pointer` to the `struct`, then the `tree_comp()` will go into an unending loop which checking such a structure. The other possibility is that the *pointer* type constructor just records the *name* of the type to which it is pointing, as shown in following figure.



This means that we use named equivalence. Many languages which allow such recursive type definitions, including C, settle for a compromise solution. They use structural equivalence in all the cases except while comparing pointers within `struct`. If we want to use structural equivalence in all the cases, including the pointers in `struct`, then we have to somehow detect the definition loop. How this is to be done is left as an exercise for the readers.

6.5 Type Inference

Type inference refers to the ability to deduce automatically the type of a value in a programming language. It is a feature present in some strongly statically typed languages. It is often characteristic of, but not limited to, functional programming languages in general. Some languages that include type inference are Visual Basic (starting with version 9.0), C# (starting with version 3.0), Clean, Haskell, ML, OCaml, Scala. This feature is also planned for C++0x and Perl 6. The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while still permitting type checking.

Let us take an example.

```
int addone(int x) {
    int result;
    result = x + 1;
    return result;
}
```

This function returns an `int`, so we can infer that `result` should be type `int`. As the `result` is obtained by adding 1 (an integer) to `x`, then we can surmise that `x` is type `int`. It will be nice if we could write the above program as:

```
int addone(x) {
    result = x + 1;
    return result;
}
```

Thus we can “reverse engineer” the types of various program components.

However, languages that support type inference to the degree the above example illustrates rarely support such implicit type conversions. Such a situation shows the difference between type inference, which does not involve type conversion, and implicit type conversion, which forces data to a different data type, often without restrictions.

Type inference is the ability to automatically deduce, either partially or fully, the type of an expression at compile time. The compiler is often able to infer the type of a variable or the type signature of a function, without explicit type annotations having been given. In many cases, it is possible to omit type annotations from a program completely if the type inference system is robust enough, or the program or language is simple enough.

To obtain the information required to infer the type of an expression, the compiler either gathers this information throughout the program as a whole and subsequent reduction of the type annotations given for its subexpressions, or through an implicit understanding of the type of various atomic values, for example, True is type *bool,* 42 is type *integer,* 3.14159 is type *float* or *double,* etc. It is through reduction of expressions to implicitly typed atomic values that the compiler for a type inferring language is able to compile a program completely without type annotations. In the case of languages having complex structure, operator overloading and polymorphism, it is not always possible for the compiler to do such inferences and hence type annotations are occasionally necessary for disambiguation of types.

6.5.1 Formal Semantics

The semantic information associated with an identifier is sometimes called its *environment*. For an identifier `a`, it is written as $E(a)$. As suggested in Exercise 1, it can be stored in the Symbol Table. As we shall see in Chapter 7, block structured languages use Symbol Table which changes its contents at run-time. Thus, the environment associated with an identifier may go on changing as the computation proceeds.

Formal Semantics is a mathematical method of defining the semantics of language constructs. Inference rules are widely used in formal specifications of programming language semantics. An inference rule defines for certain set of hypotheses what should be the conclusion and looks like this:

$$\frac{\text{Hypothesis}_1 \ldots \text{Hypothesis}_n}{\text{Conclusion}}$$

You should read this as a rule: "if all of the hypotheses 1 to *n* are true, then the conclusion will hold".

For example, the rule for type checking multiplication is:

$$\frac{E \vdash e_1 : int, E \vdash e_2 : int}{Ee_1 * e_2 : int}$$

$E \vdash e : t$ means that, in environment *E,* the expression *e* has type *t.*

To type check the occurrence of a variable, we have to check that the variable is defined in the current environment, and find the corresponding type:

$$\frac{E(v)t}{E \vdash v : t}$$

Here are some example rules:

$$\frac{}{E \vdash INTEGER : int}$$

This rule is an *axiom,* because it does not have a precedence hypothesis.

$$\frac{}{E \vdash true : boolean}$$

This is another axiom.

$$\frac{E \vdash e_1 : t \; E \vdash e_2 : t \; t \in \{int, \; boolean\}}{E \vdash \; e_1 e_2 : boolean}$$

$$\frac{E \vdash e_1 : boolean E \vdash e_2 : boolean}{E \vdash e_1 \&\& e_2 : boolean}$$

We can combine several rules to build a complete "rule tree".

$$\frac{\dfrac{E(x)int}{E \vdash x{:}int} \quad \dfrac{E \vdash 2{:}int \quad \dfrac{E(y)int}{E \vdash y{:}int}}{E \vdash 2{*}y{:}int}}{E \vdash \; x2 * y : boolean}$$

This rule tree says that in any environment where *x* and *y* are both defined with type int, the expression *x2 * y* has type boolean.

We can check statements using similar idea:

$$\frac{E \vdash \; e : boolean E \vdash s_1 E \vdash s_2}{E \vdash \; IFeTHEN \, s_1 ELSE \, s_2}$$

Note that sentences do have a type, so we just write $E \vdash s$ to indicate that *s* is well formed in environment *E*.

A function body having a statement *s*, introducing a variable *v* of type *t*, would be type-checked as:

$$\frac{\{(v, \; t)\} \oplus E \vdash s}{E \vdash tv; \, s}$$

The symbol $\oplus$ indicates environment *E* merged with variable-type pair (*v*, *t*).

Checking Variable Declarations: Formal arguments count as local variables as far as type-checking is concerned.

6.6 Type Conversion and Coercion

In some languages like C, it is possible to convert the type of a program component from one type to another. This is called *casting* in C and is written as, for example:

```
int i;
float a;
a = (float) sin((double)i);
```

Here the sine function is defined as `double sin(double)`, we cast our argument `i` to type `double`, so that the compiler is able to generate the type conversion code to be used at the run-time. Again, the variable on the LHS is type `float` and we have have cast the RHS in that type. In fact, some of such conversions, especially in assignments, are very common and do not require explicitly mentioned type casting. Thus, we could have written the above code segment as:

```
int  i;
float  a;
a = sin((double)i);
```

This is called type *coercion.* Further, we may even drop the cast in the call argument:

```
int i;
float a;
a = sin(i);
```

This is called type *promotion,* i.e., `i` of type `integer` is promoted to type double. In both these cases, implicit type casting is involved and the compiler must detect it and insert appropriate type conversion code in the compiled code.

6.7 Overloading of Operators and Functions

Operator overloading, so popular in C++, is a way to implement polymorphism for some of the unary and binary operators in a language. The overloading operates at two levels. For example, consider the four arithmetic operators +, −, * and /. Even in C language these operators are overloaded in the sense that though for both integers and floats the meaning of '+' operator is "addition of numerical value", the actual details of the operation, exceptional condition, range of valid results, etc. are quite different for them.

Though not available, we can have further overloading of '+' in some hypothetical language, for concatenation of strings, i.e., "hello" + "world" will produce "hello world". This is almost intuitive and quite useful. Further, we can overload the '*' operator for strings, such that "world"*3 will give "world-world-world". This overloading seems reasonable because a string is isomorphic to an integer – if we concatenate two strings their lengths get added.

If we consider a slightly more esoteric domain, the '+' operator can be overloaded for matrices also, as it is done in matrix manipulation languages like MATLAB, R, PDL, etc. We can even contrive to assign a reasonable meaning to the '−' operator.

Such operator overloading becomes our concern here, as it can complicate the type checking. Suppose we write `A + 5` where `A` is $m \times n$ matrix. How do we interpret it? In specialized programming languages like MATLAB, the interpretation could be "add value 5 to each element of A". However, how does the compiler know that the programmer has not made a mistake? Should it flag this construct as an error? or as a warning?

6.8 Example: Type Checking in an Interpreter

As an interpreter has to immediately execute a given program, it should be able to do some preliminary type checking. We take an example of a simple interpretive language. Its grammar is:

```
program  :  funcs END;
funcs    :  func
         |  func funcs
         ;
func     :  typeid '(' typeids ')' ':' expr ;
typeid   :  INT ID
         |  BOOL ID
         ;
typeids  :  typeid
         |  typeid ',' typeids
         ;
```

```
expr   :  NUM
       |  ID
       |  expr '+' expr
       |  expr '~' expr
       |  IF expr THEN expr ELSE expr END
       |  ID '(' exprs ')'
       |  LET ID '=' expr IN expr END
       ;
exprs  :  expr
       |  expr ',' exprs
       ;
```

It is a small *functional* language, where the program is made up of function definitions. Each function declares its result type and the types and names of its arguments. Functions and variables have separate name spaces. Though the grammar is ambiguous, we will assume that the ambiguities have been resolved during parsing.

The body of a function is an expression. Comparison operation ' ' is defined both on booleans and integers, but addition '+' only on integers. Numbers, NUM, are integers only. The tokens INT, BOOL, ID, NUM, IF, THEN, ELSE, LET, IN are returned by the Scanner on scanning appropriate lexical representations.

Since a value can be either integer or boolean, the interpreter uses a value type that contains both integer and boolean. The interpreter calls a function `error()` if any error, including type error, is found.

When an expression is to be evaluated, in addition to the AST of the expression, we need a Symbol Table `vtable` which binds a variable to its value. Also, in order to handle function calls, we need another Symbol Table `ftable` which binds a function name to the AST of its declaration. Incidently, this method of handling function definitions is similar to the way it is implemented for `hoc6` in "UNIX Programming Environment" by Kernighan and Pike. The result of evaluating an expression is the value of the expression.

We assume that the following functions are available:

getname(ID) extracts the name of an identifier from `vtable`.
getvalue() returns the value of a number.
eval(expr, vtable, ftable) returns value – either a boolean or an integer.
call(func) handles function calls.
lookup(table, name) lookup the value of a name.
bind(table, name, value) binds 'value' to 'name' in the 'table'.

We are not interested here in full implementation of the interpreter, but only in how type checking is applied. In view of that, we give various actions taken by the evaluator function `eval()` in the form of a table:

```
eval(expr, vtable, ftable) = case expr of
NUM : getvalue(NUM)
ID  : v = lookup(vtable, getname(ID))
      if v = unbound then error()
      else v
expr1 + expr2 : v1 = eval(expr1, vtable, ftable)
                v2 = eval(expr2, vtable, ftable)
```

```
                if v1 and v2 are integers then v1 + v2
                else error()
expr1 ~ expr2 : v1 = eval(expr1, vtable, ftable)
                v2 = eval(expr2, vtable, ftable)
                if v1 and v2 are both integers or both booleans
                   then if v1 = v2 then True else False
                else error()
IF expr1 THEN expr2 ELSE expr3 END:
                v1 = eval(expr1, vtable, ftable)
                if v1 is a boolean then
                   if v1 = True then eval(expr2, vtable, ftable)
                                 else eval(expr3, vtable, ftable)
                else  error()
ID(exprs) : def = lookup(ftable, getname(ID))
            if def = unbound then error()
            else args = evals(exprs, vtable, ftable)
                 call(def, args, ftable)
LET ID = expr1  IN expr2 END: v1 = eval(expr1, vtable, ftable)
                        vtable' = bind(vtable, getname(ID), v1)
                        eval(expr2, vtable', ftable)
endcase
```

The function `evals` builds a list of the values of the expressions in the expression list. A list is written in square brackets with commas between the elements. The operator '::' adds an element to the front of a list.

```
evals(exprs, vtable, ftable) = case exprs of
expr : '[' eva1(expr, vtable, ftable) ']'
expr ',' exprs : eval(expr, vtable, ftable) '::' evals(exprs, vtable, ftable)
```

Note that *all the if predicates on the RHS in the above table are concerned with type checking*. For example, interpreting a function call will proceed as follows. A function declaration explicitly declares the types of the arguments. When a function is called, it must be checked that the number of arguments is the same as the declared number, and that the values of the arguments match the declared types.

If this is true then we build a Symbol Table that binds the parameter variables to the values of the arguments and use this in evaluating the body of the function. The value of the body must match the declared result type of the function. So the `call()` is to be implemented as:

```
call(func, args, ftable) = case func of
typeid '(' typeids ')' '=' expr : (f,t0) = gettypeid(typeid)
                                  vtable = bindtype(typeids, args)
                                   v1 = eval(expr, vtable, ftable)
                                   if type(v1) = t0 then v1
                                   else error()
endcase
```

The function `gettypeid()` just returns a pair of the declared name and type.

```
gettypeid(typeid) = case typeid of
INT ID : (getname(ID), INT)
```

```
BOOL ID : (getname(ID), BOOL)
endcase
```

The function `bindtype()` checks the declared type against a value and builds a Symbol Table that binds the name to the value if they match. It also checks if all parameters have different names by checking il the current name is already bound.

```
bindtype(typeids, args) = case(typeids, args) of
'('typeid ','']' v ']'')'  :  (x,t) = gettype(typeid)
                              if type(v) = t then bind(emptytable, x, v)
                              else error()
'('typeid','typeids','
'('v'::'vs')'')'           :  (x,t) = gettype(typeid)
                              vtable = bindtype(typeids, vs)
                              if lookup(vtable, x) = unbound and type(v) = t
                               then bind(vtable, x, v)
                              else error()
anything else              :  error()
```

A program is executed by calling the `main()` function, which must be present in any program, with a single argument that is the input to the program. So we build the Symbol Table for the functions, look up `main()` in this table and call `call()` with the resulting definition and an argument list containing just the input.

```
run(program, input) = case program of
funcs : ftable = buildftable(funcs)
        def = lookup(ftable, 'main')
        if def = unbound then error()
        else call(def, [input], ftable)

buildftable(funcs) = case funcs of
func  : f = getfname(func)
        bind(emptytable, f, func)
func funcs : f = getfname(func)
             ftable = buildftable(funcs)
             if lookup(ftable, f) = unbound
                then bind(ftable, f, fcun)
             else error()

getfname(func) = case func of
typeid'('typeids')''=' expr : (f,t0) = gettype(typeid)
                              f
```

Note that you can implement this interpreter by using `yacc` to parse the grammar and `yacc` *actions* to specify the activity on the RHS of the above tables, when the corresponding syntactic construct given on the LHS is parsed. You are told to implement a limited portion of this interpreter, with all its type checking in one of the Exercises. Here we give some hints about how type checking can be done.

We shall assume the use of the Node data structure given below:

```
typedef struct node_struct{
struct node_struct *link[2];
util u,v,w,x,y,z;
}Node;

typedef Node Symbol;
#define L(n)       ((n)->link[0])
#define R(n)       ((n)->link[1])
#define name(n)    ((n)->w.S)
#define type(n)    ((n)->v.I)
#define ival(n)    ((n)->u.I)
#define ttree(n)   ((n)->z.N)
#define subtype(n) ((n)->y.I)
```

Note that while type checking identifiers and numerical constants, we have to deal with two levels of types – syntactic types like ID, NUMB, IF, THEN, etc. – and semantic value types like INTEGER, BOOLEAN, FLOAT, etc. Thus an identifier has *type* ID and *subtype* INTEGER or BOOLEAN. We have provided for this requirement in the Node structure. Also, with each syntactic token, we have to associate its constructor tree as discussed in Section 6.2. This tree is linked via the link field `ttree`. Further, note that type `Symbol` is made equivalent to type `Node`.

The grammar as given above is ambiguous and we need to take help of operator precedences so that `yacc` can resolve the ambiguities adequately. The following precedence specification seems reasonable:

```
%union {
   Symbol *sym; /* symbol table pointer */
}
%token <sym> INT BOOL ID NUM IF THEN ELSE LET IN END UNDEF
%token   ':' ';'
%left    ','
%token   '='
%left    '~'
%left    '+'
```

We now show `yacc` actions for some of the `expr` constructs. You should compare them against the table given above.

```
expr  : NUM { $$ = copy_node($1); }
      | ID  { int t; char s[40];
              if((t = type($1) == UNDEF){
                  sprintf(s, "undefined [%s]\n",name($1)); yyerror(s);}
              else { $$ = copy_node($1);}
            }
      | expr '+' expr { int x,y; char s[40]; x = ival($1); y = ival($3);
                        if(subtype($1) == INT && subtype($3) == INT){
```

```
                    $$ = tree_node(gentemp("plus"));
                    ival($$) = x + y;
                    subtype($$) = subtype($1); type($$) = subtype($1);
                    subtype(install(name($$),subtype($$),ival($$)))
                                = subtype($1);}
                else{sprintf(s,"not int [%s %s]\n",name($1),
                     name($3)); yyerror(s);}
                }
```

In the above, `tree_node()` is a utility function which generates a new Node with a given name attribute. Unique names with specified prefix are generated by `gentemp()`.

We have given below one more portion of the evaluator – `LET` construct which substitutes a value for an identifier in an expression.

```
| LET ID '=' expr { Symbol *sp; sp = lookup(name($2)); type(sp) = ID;
                    subtype(sp) = subtype($4); ival(sp) = ival($4); }
                   IN expr END %prec ';'
                  { type($2) = ID; subtype($2) = subtype($4);
                    ival($2) = ival($4); $$ = tree_node(gentemp("letexpr"));
                    ival($$) = ival($7); subtype($$) = subtype($7);
                    type($$) = type($7);
                    subtype(install(name($$),subtype($$),ival($$))) =
                                                          subtype($$);
                  }
;
```

To properly implement the semantics of the `LET` construct, it is required to update the attributes of the `IDentifier` *in the Symbol Table* with values obtained from the `expr`, *before* the second `expr` is computed. This requires two actions – first updates the attributes of the `IDentifier` and the second computes the second expression and assigns its value to the LHS and inserts in the Symbol Table for diagnostics purpose.

? EXERCISES

1. What are the primitive types in C and Java?
2. Enumerate all the type coercion and promotion rules in C.
3. Investigate the types as used in Perl language.
4. How will you implement the definition loop checking suggested in Section 6.4.1?
5. Implement the `expr` evaluation portion of the simple functional language interpreter given in Section 6.8, with all its type checking. The grammar is already given in `yacc` format. You will have to write `yylex()`, symbol tables manipulation functions (steal our tree-based functions). Remember that the emphasis is on type checking.

WEB RESOURCES

Latest version of GNU Bison (2.4.3 – 05-Aug-2010) is available at:
`http://www.gnu.org/software/bison/`

On-line documentation:
`http://www.gnu.org/software/bison/manual/html_node/index.html`

FURTHER READING

1. Kenneth C. Louden, "Compiler Construction – Principles and Practice", Thomson. Chapter 6.
2. Aho, Lam, Sethi and Ullman, "Compilers – Principles, Techniques and Tools", Pearson Education. Section 6.5.

GLOSSARY

Type Signature of a function consists of its return type plus type of each of its formal arguments. The compiler uses type signature to validate proper invocation of a function.

7

Run-Time Environment

What you will learn in this chapter

- What is a Run-Time environment for a program?
- What is the basic CPU and Memory environment?
- What are different methods of storage allocation?
- What is an Activation Record?
- What are different types of Activation Records?
- What are different methods of passing parameters to a subroutine?
- How is memory allocation managed for a block-structured language?
- How are the scope and visibility of a variable determined?
- How is a running program controlled by an Operating System?
- What are System Calls?
- What is a Language library?
- How does one create an application-specific library?
- What are system environmental parameters?
- How are the command-line invocation parameters passed?

Key Words

run-time, memory organization, static, stack-based and dynamic run-times, activation record, parameter passing, heap allocation, block-structured languages, variable scope and visibility, process, system calls, language library, system environmental parameters, command-line parameters

We have been discussing till now the *front-end* of a compiler and something about how semantic analysis, i.e. finding the "meaning", of a program is done. We also considered certain essential semantic checks, known as `type checking`. We have to now consider how the target language code is generated, but before that we should know the environment which our program experience when it is running. The details of generated code depend heavily on this environment, although the general principles of code generation steps are the same. Some of the code optimization that a good compiler attempts also depend upon the run-time environment. That is why we discuss in this chapter the *run-time environment* – the background canvass on which the picture of running programs will be painted.

One should really consider the total run-time environment under which a program will have to run. For example, the environments under which a program running on a desktop computer, an application

running on a personal digital assistant (PDA), an application being executed on a cloud computer or a program running on a supercomputer are quite different. Figure 7.1 shows the overall environment in which a translated program executes. The total environment consists of the following:

Hardware: It includes CPU, memory, I/O devices, communication network, etc. It – especially the processor and the memory – provides the basic environment, on which we will concentrate.

Operating system: Concerned with resource management. It provides services through System Call Interface (SCI).

Language library: It provides functions for doing the jobs which the basic hardware is not directly capable of doing. For example, in C language if we write as

```
short i;
int j;
j = j + i ;
```

we expect that the `short` value on the RHS will be *promoted* to `int` and assigned to `j`. On machines where this cannot be done via an instruction, the compiler has to generate a call to a C language library to do this work. A slightly more involved example is the *cast* facility provided in C. On machines which do not have floating-point processor, the library will be used to do the floating-point computations.

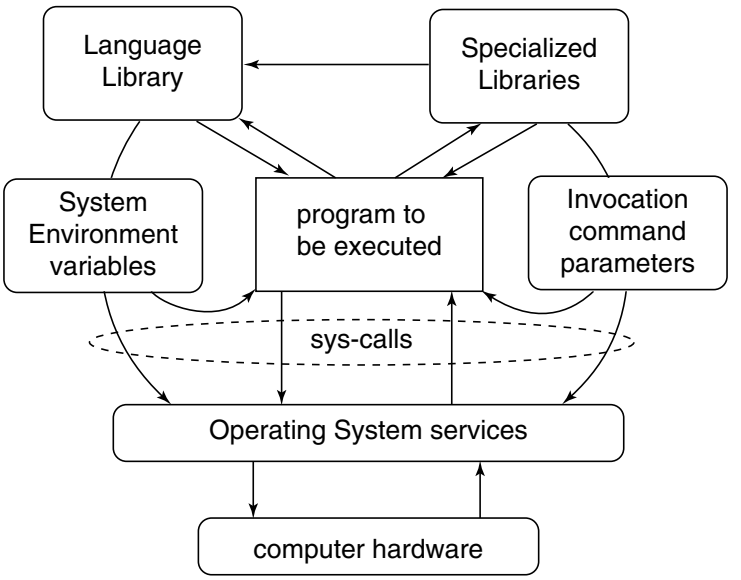


Fig. 7.1 Run-time environment of a translated program, apart from the basic environment that is the CPU

System environmental parameters: System-wide and user-specific environmental values, such as – absolute paths to essential software (compiler, libraries, utilities), Terminal type, users home directory, user name, user group, default permissions for newly created files, prompt strings, machine type, O/S version, last console command executed, etc. These parameters are available in a C program by a third argument in the `main()`, like:

```
int main(int argc, char *argv[], char *env[]){ ... }
```

Also, C library functions `getenv()` and `putenv()` can be used to obtain and set value of an individual environment parameter.

Invocation (command-line) parameters: The command-line arguments which are accessible via the second argument in the `main()`.

Specialized libraries: A particular program may require and depend upon a special functions library, for example graphics library.

The most basic environment – the CPU plus memory pair – is not shown in Fig. 7.1 separately, but we concentrate on it more, as it is immediately related to our language translation tasks.

As an aside, we note that the source of power and thermal states is also part of the total environment, especially in portable equipment, but we simply ignore them here. However, there are situations where a program or application to be run on an embedded or portable system needs to be adjusted for minimum power consumption or thermal output.

Processor and memory technology has evolved over the years and their evolution has enabled computer scientists and engineers to provide more and more sophisticated language facilities. Figure 7.2 shows a three-dimensional space of processor (*P*), memory (*M*) and instruction set (*I*) in which past and present day CPU hardware exists.

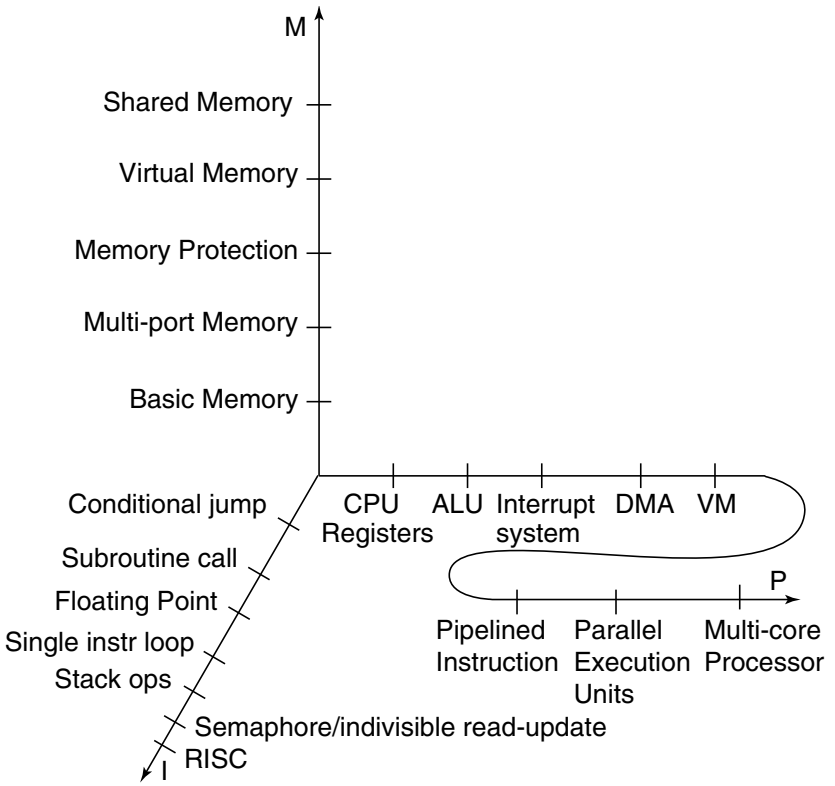


Fig. 7.2 Growth of the three dimensions of the basic environment – processor facilities, memory sophistication and instruction set capabilities

For example, it was the availability of hardware stack which allowed economic implementation of recursive functions. Virtual memory (VM) made the programmer free from program and data size limitations for normal programs.

It also provided much simpler target language generation. Interrupt system, DMA, virtual memory, indivisible read/update instructions and memory protection allowed building reliable multi-tasking environments.

Out of all the environmental factors which happen to be a resource, one that is always in short supply is the main memory. The amount of memory available and the way it is used in a system determine not only the sophistication of the language that can be used to program the system, but also the efficiency of execution. Hence, we start our discussion with the memory allocation during run-time.

7.1 Run-Time Storage Allocation

Before we discuss Code Generation phase of a compiler, we should be aware of how the code and data are stored in the memory of the computer at run-time. In case of embedded systems, where the computer is a part of a complete device, instrument or system, the program resides permanently in a non-volatile memory.

For small, one-time applications a load-and-go scheme shown in Fig. 7.3 may be used, where the source code editor, compiler and/or interpreter, the run-time library and the user's application program – both its source code version and translated version – all reside in the main memory of the computer.

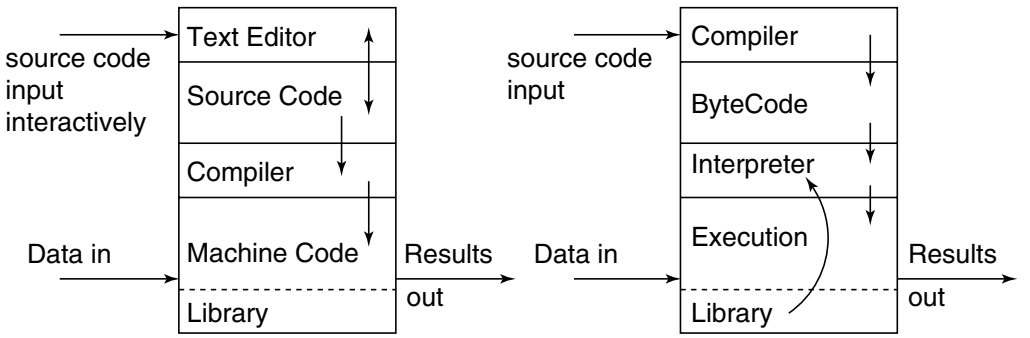


Fig. 7.3 Load-and-Go scheme. Left-hand side shows compiler-based (e.g. Turbo-Pascal) and right-hand side the interpreter-based (e.g. Perl) schemes

For most of the general-purpose computing requirements, the final *executable* is stored on an external storage medium like a hard disk and loaded in the main memory at execution time. While being loaded, the user's executable is linked with the language and other libraries. You may like to consult the description of Linker and Loader operations in Appendix C to understand how this final step before the program execution is done.

The memory manager of the Operating System has to allocate memory for the program (strictly speaking, the process). Over the years various methods of memory allocation have developed to cater for diverse needs, limited by available resources. These allocation methods have also influenced the High Level programming language design. The following are the run-time storage allocation methods:

Static storage allocation: Also called *Compile-time* storage allocation. This method is adequate for languages with statically defined data structures, for example FORTRAN.

Dynamic storage allocation: Also called *Run-time* storage allocation. This method is required and used for languages with dynamically defined data structures and recursive subroutines, for example PASCAL, C, ALGOL, C++, Java.

Explicit or implicit heap storage allocation: Also called *On-demand* storage allocation. This method of storage allocation is required and used by languages in which execution-time storage allocation requests by the program are allowed. The demand for the storage allocation may be explicit – like `malloc()` in C, or implicit – like growing strings in PL/I or LISP, where the programmer does not write an explicit statement for memory demand, but some language construct results in such demand.

We shall now discuss each of these methods in some details.

7.1.1 Static Allocation

The basic characteristics of this method of storage allocation are that the Run-Time (relative) address of each data and code object be known as the *Compile-time*, and these addresses are not expected to change during the program loading or execution. To ensure this requirement, the following two conditions must be satisfied:

1. The size of each data object is known at compile-time. This means no variable length strings, no dynamic arrays.
2. Only one occurrence of each object is allowed at any instant at run-time. This means no recursive procedures or functions, no nested procedure definitions.

A typical language which meets these constraints is FORTRAN and thus the static storage allocation is suffice for FORTRAN.

This method is simple to implement. The Scanner prepares a Symbol Table entry for each identifier, and as the type and the size of each object are known, an address relative to the beginning of the data area can be immediately allocated by the Scanner itself.

Further, we may have two kinds of run-time environments for such simple memory allocation scheme:

Single job at a time: The computer will have only one program, corresponding to a single job, at a time in its memory. This is the simplest situation from the viewpoint of the run-time environment. The computer is either running the single program loaded or is idling. In this case, the Scanner can assign *absolute* addresses to various data objects and later the Parser can assign fixed locations to the code objects. This is called *Early Binding* of addresses.

Multiple jobs at a time: The computer will be loaded with several jobs (programs) at a time and the system switches between programs when the currently running program has to wait for sufficiently long time, for example rewind of magnetic tapes, long printouts, etc. A famous operating system of this type was IBM Disk Operating System (DOS) (not to be confused with Microsoft DOS, which did not have this multi-programming capability, at least in its initial versions). In a way, such a system is a forerunner of modern multi-tasking operating systems.

To support such operation, the Scanner will have to assign *Relative* addresses, with respect to the start of the data area, which is generally considered zero. The system loader program will decide the actual load address for the data (and code) areas. This is called *Late Binding* of addresses. The loader

may have to adjust some address-dependent data, for example address constants and code addresses within some instructions, depending upon the nature of the instruction set of the computer.

If the compiler allows separate compilation of source files constituting a job, one more intermediate step of linking is involved. The linker will collect the data areas of all the constituent object modules (files) together into one data area and the loader will assign a single load address to it. Figure 7.4 shows these operations. Note that it is not desirable that the program code is disturbed or changed in anyway while the program is running (though in the past writing programs with clever self-modification was considered a challenge) and thus the code area should have protection which will make it Read-only for the users and read–write for the O/S function (otherwise you cannot load the program).

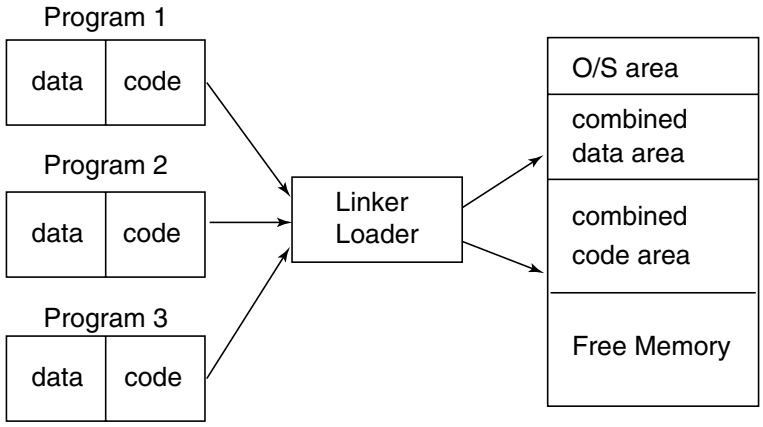


Fig. 7.4 Linker combines the object modules; loader assigns final addresses and loads them

One important portion of the data area for a procedure is the place where administrative information about the invocation of that procedure is stored. A typical layout of such allocated data area for each module or routine is shown in Fig. 7.5. Such an area is generally called an Activation Record (AR). When the full AR is stored on a stack, it is also called a *stack frame*, though we should make a distinction between AR which is a logical concept and the stack frame which is an actual storage area for it.

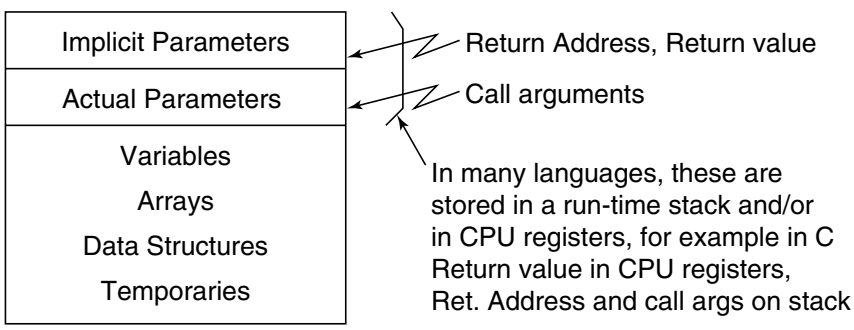


Fig. 7.5 Static memory allocation: data area per module, or Activation Record

We may further note that in case of RISC processors, which generally have a large register file, consisting of say 32, 64 or even more full-length registers, the whole of the current AR may be kept in some of these registers for fast access. Even in CISC processors, special purpose registers like Program Counter (PC) or Instruction Pointer (IP), Stack Pointer (SP), Frame Pointer (called Base Pointer on Intel x86 machines) are used to store some of the fields of the current AR.

An important aspect of the Run-Time environment is deciding the protocol for invoking a subroutine. The following are included in such a protocol:

- call sequence: allocation of storage area for the AR;
- call sequence: allocation of storage space and access to the call arguments;
- call sequence: saving and loading required CPU registers for use during the subroutine execution;
- return sequence: putting the returned value, if any, in a place accessible to the caller;
- return sequence: restoring and adjustment of CPU registers;
- finally return back to the caller.

Some of the steps given above can be done either by the caller or by the called routine and deciding in the call protocol design which is done by which of them is not an easy issue to resolve. The call sequence and return sequence together are called the *Calling Conventions* and we shall discuss them at some length later.

Fully Static Run-Time

By fully static run-time we mean one without the use of even a single stack. Such an environment is suitable for a language like FORTRAN, BASIC or COBOL. As all variables remain fixed in memory throughout the program execution time, there is a single AR for each subroutine and all such ARs are kept in the data at fixed locations. The calling convention is:

1. call: compute and copy each argument in its allotted space within the called routine's AR;
2. call: save the return address in space allotted to it in the AR (on some later machines, a special CPU register called LINK was used to store this value);
3. call: jump to the start address of the called routine;
4. routine: computes and places the return result in its allotted place, either in AR or in a CPU register;
5. return: jump to the return address stored in AR (or special register);
6. Many CPU architectures have special subroutine *call* and *ret* instructions which save the return address and jump to the saved return address, respectively.

You can immediately see why such a memory allocation and calling convention cannot handle recursive or mutually recursive subroutine calls. Once a subroutine is called, its AR is loaded with the return address, which would be destroyed by any subsequent call, which is however needed for proper return to the caller. Sorry, you cannot implement on such a system the recursive functions for factorial or Fibonacci series computation that you are so familiar with.

It is instructive to discuss, before the Dynamic memory allocation methods, how a typical stack-based memory allocation code works, especially with regard to the subroutine invocation protocol.

7.1.2 Typical Function Calls Interface for C

As a preparation before we discuss the *Dynamic Memory allocation*, we first discuss function calls for a very simple memory model, called small memory model used in a legacy C compiler (Borland C). The small memory model means the Code and Data segments are limited to 64 kbytes each and the

Stack is included in the Data segment. With this memory organization, 16-bit (2-bytes) addresses are suffice.

Function call and parameter passing: Consider a function call `test(i, j, 1);`. This call results in the following actions:

1. Call arguments are pushed in the right-to-left order.
2. Restore the stack after the call is returned. This means that the calling function is responsible for stack clean-up. This is called *Caller clean-up* method of calling convention.
3. The returned value, if 16-bit then via register AX, if 4-bytes then via DX + AX and if a float then via top-of-stack, ST(0), in floating-point processor.

See Fig. 7.6(a). The relevant segment of the caller routine is:

```
movw   1, %ax
pushw  %ax
pushw  j
pushw  i
call   test
addw   6, %sp       // stack clean-up
```

Function's handling of arguments, local data: Referring to Fig. 7.6(b), immediately on entry, the called function `test()` pushes BP and copies SP to BP. Now BP is the `base` for all the

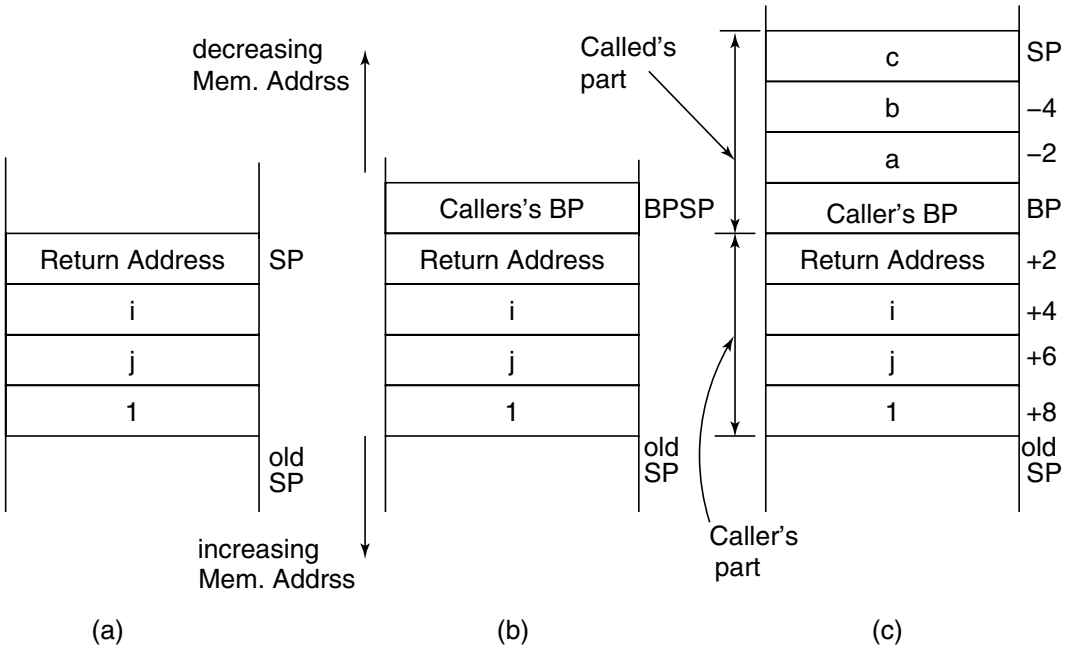


Fig. 7.6 Function call management in a legacy C compiler. The stack (a) just after the call is executed, (b) just after entry to the called function and (c) while the called function is executing. After the return from the called function, SP is restored to "old SP" by the caller

addressing within the called function. The call arguments are available at (BP) + 4, (BP) + 6 and (BP) + 8. The return address is at (BP) + 2 and the caller's BP is at (BP).

The actions within the called function are:

1. Push BP: the caller's BP is saved in the stack;
2. Copy SP to BP: set up the current frame (base) pointer;
3. Do whatever computation for this function, but do not disturb BP;
4. Pop BP, the caller's BP restored;
5. Return from this function. Note that the return address will be at TOS.

The relevant code in function `test()`, assuming that it does not use any local variables, is:

```
test:
        pushw  %bp
        movw   %sp, %bp        // figure (b)
        movw   4(%bp), %ax     // i
        addw   6(%bp), %ax     // j
        subw   8(%bp), %ax     // 1
        popw   %bp             // caller's BP restored
        ret
```

Suppose the called function `test()` used three local variables, `a`, `b` and `c` for its computation, in that case the initial instructions will be slightly different and the code will be:

```
test:
        pushw  %bp
        movw   %sp, %bp
        subw   6, %sp          // allocate locals, figure (c)
        movw   4(%bp), %ax     // i
        addw   6(%bp), %ax     // j
        subw   8(%bp), %ax     // 1
        // use locals as -2(%bp), -4(%bp) and -6(%bp)
        movw   %bp, %sp        // restore SP, deallocate locals
        popw   %bp             // caller's BP restored
        ret
```

The local (or automatic) variables `a`, `b` and `c` are accessible at (BP)-2, (BP)-4 and (BP)-6, respectively. Again note that BP must not be disturbed during the function.

Figure 7.6(c) shows the situation while `test` is computing. In short, call arguments are at address positive w.r.t BP, local variables are at address negative w.r.t BP, caller's BP is at (BP) and the return address is at (BP) + 2.

The visual shown in Fig. 7.7 may help remember the way such calling convention lays out the call parameters and local variables.

7.1.3 Dynamic Allocation

Data area requirements are not known completely at compile-time. The size and number of each data object, although not known at compile-time, must be known at run-time, on entry to the block of code in which they are defined. Thus, dynamic allocation strategy is a requirement for block-structured languages like C, PASCAL, ADA, etc.

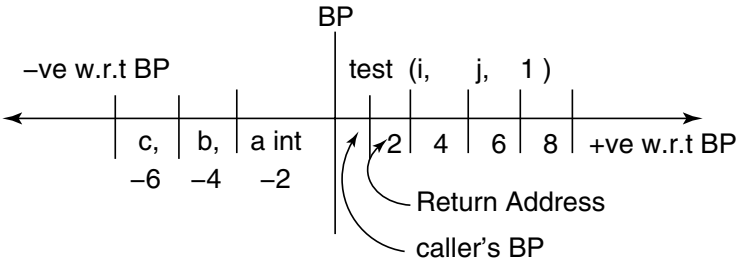


Fig. 7.7 A visualization of parameters layout with BP as the base

The dynamic memory allocation methods are of two types:

Stack based: To support run-time allocation arising out of procedure call/return activities and need for local variable;

Heap based: To support run-time on-demand allocation, arising out of explicit and/or implicit memory demand.

In real-life languages like C, all the three methods of memory allocation are used – code is allocated in a static manner and data are allocated space by static, stack- and heap-based methods, depending upon the nature of data declarations. Figure 7.8 shows typical memory allocation for a C language program.

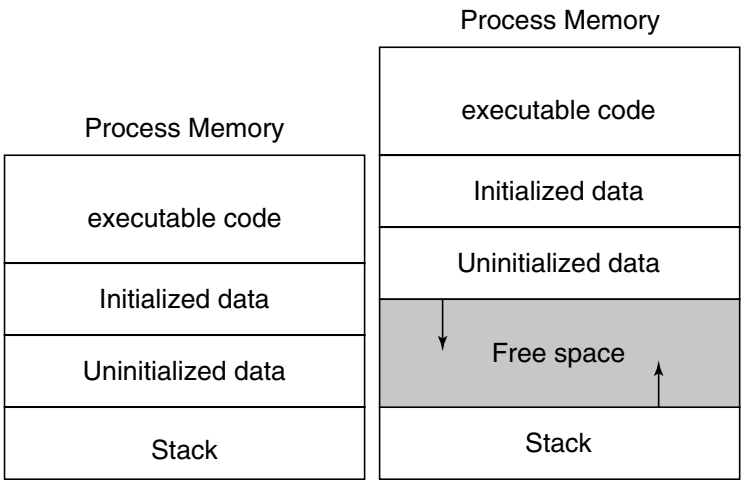


Fig. 7.8 Process memory allocation for C. Code and initialized data are allocated statically, uninitialized allocated heap-based and stack is for all stack-based allocation, i.e. ARs and any temporary variables in functions. Left-hand side: logical view, right-hand side: layout within virtual memory

Stack-based Allocation

This method is implemented using a stack called Run-Time stack. Logically, each block is having its own data area. The legacy compiler function calling convention discussed in Section 7.1.2 is an example of such an allocation method. The basic method remains the same; the only major difference

is that most of the modern CPU are 32-bit (4-byte) or 64-bit (8-byte) machines, so the address adjustments with respect to BP or SP will be in chunks of 4-bytes. Also, we shall use the extended registers `%ebp`, `%esp`, `%eax`, etc.

Languages Without Nested Procedure Definitions

We shall consider first languages like C, which do not allow nested procedure definitions.

First, we consider simplest of all cases – no call arguments, no local variables and no return value.

```
void func(){
}
int main(){
  func();
}
```

As there is no call argument or return value from the function, we cannot use this function as a recursive one. The following assembly code shows the minimal links generated for this function. The function does not *do* anything, just returns after its invocation. The two stack frame control registers, `%esp` and `%ebp`, have little to do. Initially `%ebp`, which on entry to this function is pointing at the bottom of the stack frame of the caller, is saved on the stack (it is generally called *old BP*). Now `%esp` is pointing to this save location on the stack, so `%esp` is copied into `%ebp`. This establishes the bottom of the current stack frame. Now onwards, till the exit from this function, `%ebp` will not be changed at all.

```
        .text
.globl func
        .type   func, @function
func:
        pushl   %ebp
        movl    %esp,  %ebp
#---- that was entry, now we return ---------
        popl    %ebp
        ret
        .size   func, .-func
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp      # 16-byte align
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
#---- that was entry ----------
        call    func
#---- now we return -----------
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
```

When the function returns, it restores `%ebp` to its *old BP* value, i.e. the pointer to the bottom of the caller, so that the caller will not see a different world when it gets back control. Due to the `pop`, now `%esp` is pointing to the return address, so a `ret` instruction is issued straight away. Effectively, whatever `func()` did on the stack is wiped out.

This simplest of linkage satisfies three essential conditions for a function called protocol:

1. setting up return address;
2. setting up the reference point (bottom) of the called function's stack frame, `%ebp` contains this reference;
3. saving the *old BP*, i.e. reference to the bottom of the caller routine, so that on return from the called function, the caller sees the same status of the stack.

The main function is interesting. Though it is called just as a function by the O/S, as per current conventions, the GNU C compiler `gcc` ensures that the stack frame is aligned on a 16-byte boundary, this is ensured by the `andl` instruction at the entry. However, we must remember the old non-aligned `%esp`, because that is where the Return address is. This is done via `%ecx` register (see Fig. 7.9). The 16-byte alignment is a `gcc` feature controlled by option `-mpreferred-stack-boundary = n`, where the compiler tries to keep items on the stack aligned to 2^n. If you change n to 2, it would only allocate 8-bytes on the stack. The default value for n is 4, i.e. it will try to align to 16-byte boundaries. From Pentium III onwards, the Streaming SIMD Extension (SSE) data type __m128 may not work properly if it is not 16-byte aligned.

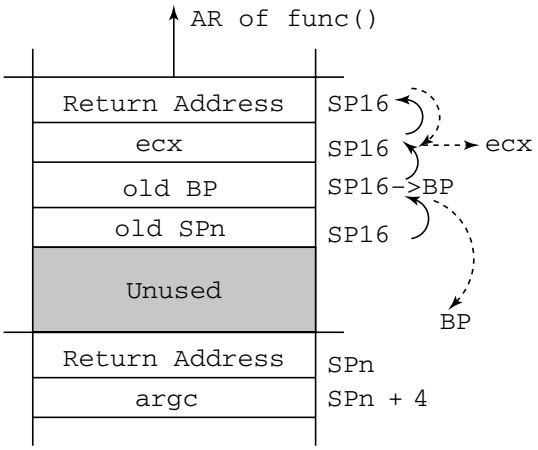


Fig. 7.9 AR of the `main()` with alignment of the SP on 16-byte boundary. SPn is the unaligned SP and SP16 is 16-byte aligned SP

Next consider a function without any call arguments or local variables.

```
int func(){
  return 1;
}
int main(){
  int r;
  r = func();
}
```

The only difference as far as the stack-based storage allocation is concerned is that the called function returns a value through `%eax` and the calling `main()` function retrieves it and stores it in place for the local variable r, at `-8(%ebp)`. In the code listing below, note that the SP is always kept aligned on 16-byte boundary.

```
        .text
.globl  func
        .type  func, @function
```

```
func:
          pushl    %ebp
          movl     %esp, %ebp
          movl     $1, %eax
          popl     %ebp
          ret
          .size    func, .-func
.globl main
          .type    main, @function
main:
          leal     4(%esp), %ecx
          andl     $-16, %esp
          pushl    -4(%ecx)
          pushl    %ebp
          movl     %esp, %ebp
          pushl    %ecx
#-------  that was entry ---------
          subl     $16, %esp     # make space for locals
          call     func
          movl     %eax, -8(%ebp)
          addl     $16, %esp     # kill locals
#-------  return -----------------
          popl     %ecx
          popl     %ebp
          leal     -4(%ecx), %esp
          ret
          .size    main, .-main
```

Consider a C function `factorial(n)`, written without use of any local variables, but there is a call argument:

```
int factorial(int n){
       if(n <= 1) return 1;
       return n * factorial(n - 1);
}
```

This recursive function when invoked by the caller routine by a call like `factorial(3)` will result in stacking of 3 ARs on the run-time stack as shown in Fig. 7.10. The assembly language form of the generated code is:

```
          .text
.globl factorial
          .type   factorial, @function
factorial:
          pushl   %ebp
          movl    %esp, %ebp
          subl    $4, %esp
          cmpl    $1, 8(%ebp)
          jg      .L2
          movl    $1, -4(%ebp)
          jmp     .L4
```

```
.L2:
        movl    8(%ebp), %eax
        decl    %eax
        subl    $12, %esp
        pushl   %eax
        call    factorial
        addl    $16, %esp
        movl    %eax, %edx
        imull   8(%ebp), %edx
        movl    %edx, -4(%ebp)
.L4:
        movl    -4(%ebp), %eax
        leave
        ret
        .size   factorial, .-factorial
```

As seen from this example, if the function being called has local variables or generates temporaries during computation, space for them is provided on the stack, at positions below `%ebp`, i.e. at `-4(%ebp)`, etc.

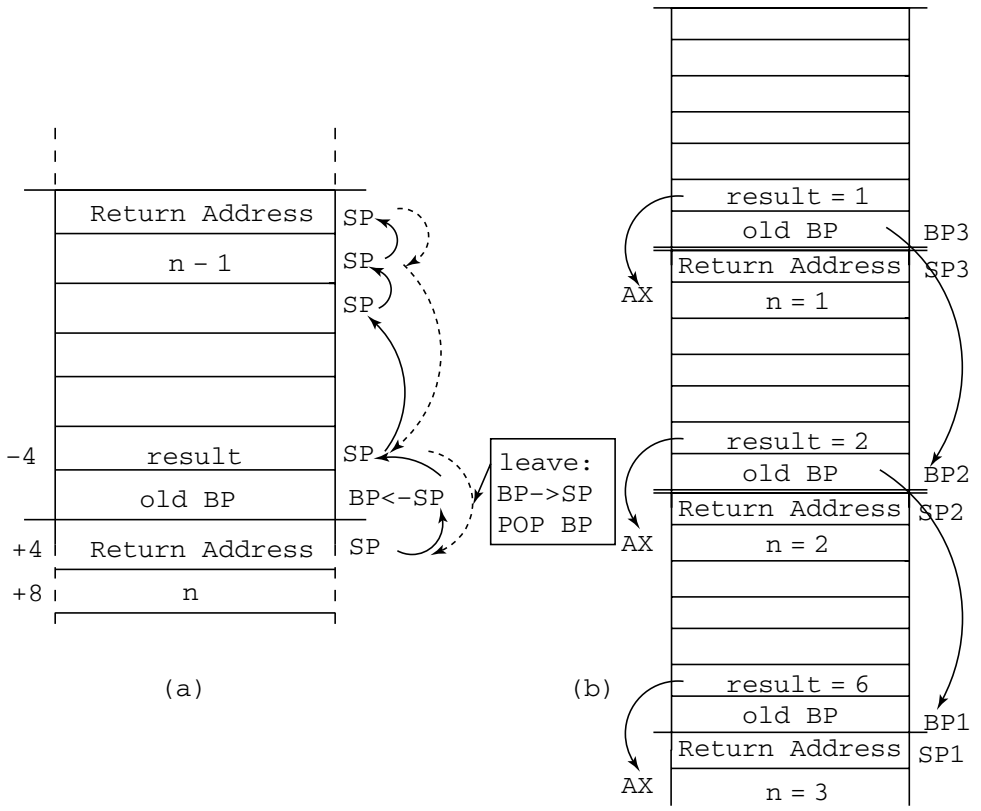


Fig. 7.10 Activation Record: no nested, no locals

Languages with Nested Procedure Definitions

When a block is entered, the required space for its AR is requested on the run-time stack (a "push" operation) and on exit from the block that area is released (a "pop" operation). Whenever the control flow is returned to the calling routine, the content of its data area must be the same, except for the returned value, as when the call was originally made. Figure 7.11 shows the manipulation of ARs for a program in a "block-structured C".

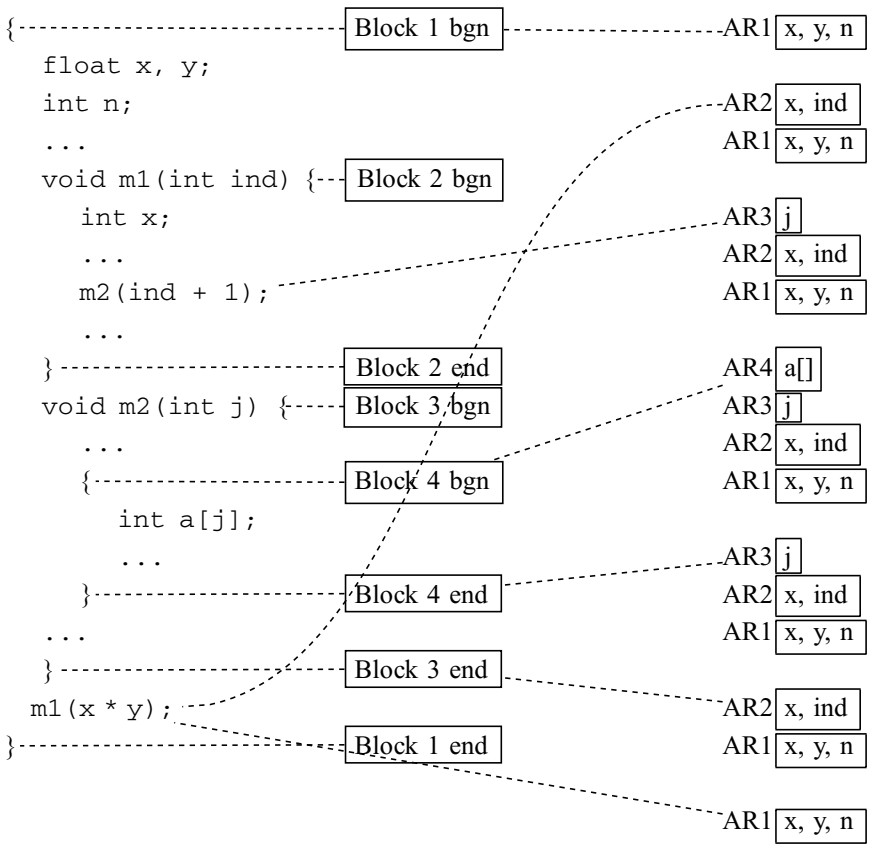


Fig. 7.11 Activation Records pushed and popped: block-structured language with nested function definitions

Activation Record (AR) – An Overall Look

We now summarize our discussion about Activation Records in the examples given above. One AR is used for each invocation of any function. An AR contains three types of data:

- **Local variables**
- **Parameters:** Both implicit and explicit:
  - **Implicit parameters:** The return address, a pointer to the base of the calling routine's AR (the so-called old BP) and return value (may not be present). Saved SP and temporary locations may also be used, depending upon the needs of the function.

- **Explicit parameters:** Either values of or address reference to the actual call arguments.
- **Static links:** Contains information necessary to access variables which are global with respect to the block currently being executed. It generally contains a number of pointers, each pointing to the base of the AR of such blocks which are global.

In case of stack-based memory allocation, the Activation Records are created on the stack.

Referring to the example in Section 7.1.3, the block static nesting structure is as shown in Fig. 7.12(a). You should carefully distinguish this static nesting from the dynamic (or run-time) nesting which is shown in Fig. 7.12(b). Variable visibility is by static or lexical nesting.

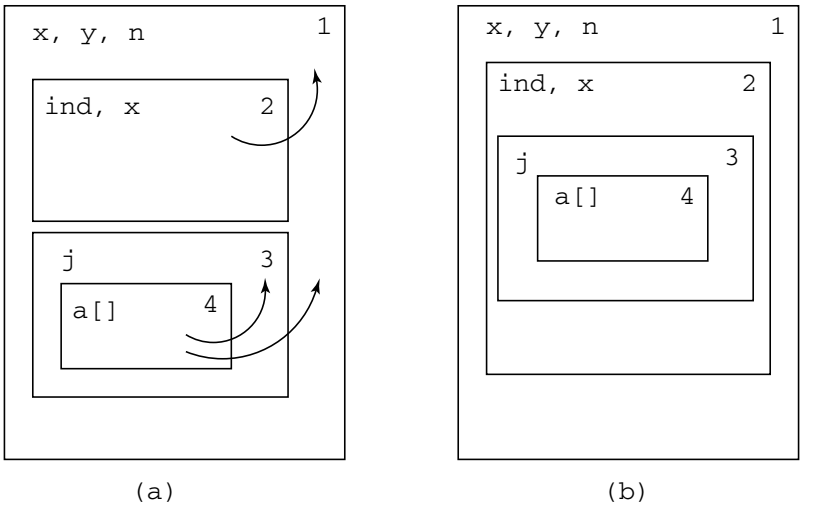


Fig. 7.12 Block nesting structures for the example. (a) Static (lexical) nesting. (b) Dynamic (run-time) nesting

For example, when Block 4 is being executed, Blocks 1 and 3 are global with respect to it and the variables therein are accessible from within Block 4. Therefore, the static link area of AR of Block 4 should contain information necessary to access these global blocks. The two kinds of links are shown in Fig. 7.13.

In case of C – a partially block-structured language – the matter is simpler. For example, for the following function

```
int g;
int func(int i){
    int l;
    return g + l;
}
```

which accesses a global variable `g`, the assembly code is

```
        .text
.globl func
        .type   func, @function
```

```
func:
        pushl   %ebp
        movl    %esp,  %ebp
        subl    $16, %esp           // space for locals
        movl    g, %eax             // get value of g
        addl    -4(%ebp), %eax      // add 1
        leave
        ret
        .size   func, .-func
        .comm   g,4,4               // here is where globals
                                    // are stored
```

In C language, the globals are universally global, while in fully block-structured languages globals are generally *locally* global.

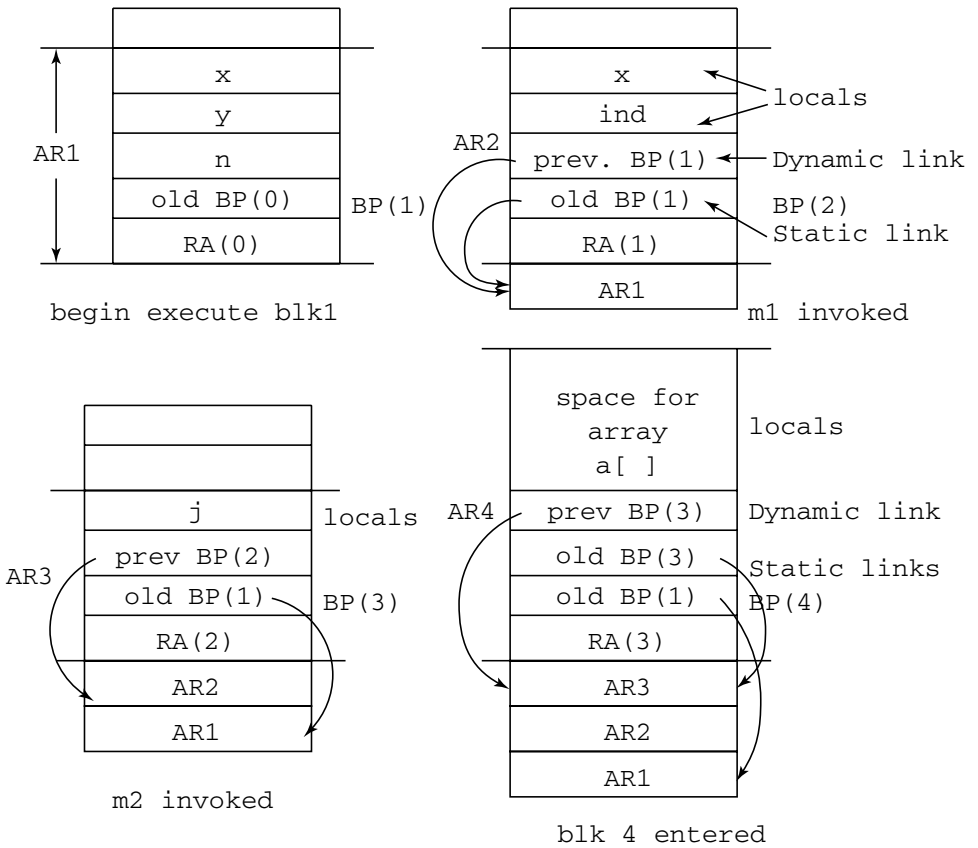


Fig. 7.13 Static and dynamic links in ARs during execution of the example program. Begin execution of Block 1: BP(0) and RA(0) possibly refer to O/S command invoking this block. Function m1 invoked: BP(1) points to BP(AR of Block 1) as it is *lexically nested* within Block 1. Prev BP(1) also points to the same BP(AR-Block 1) as it was *invoked* from Block 1, similarly for other invocations

Generally for a fully block-structured language, a two-component addressing scheme is used:

```
( <block lexical level, B>, <occurrence number, N> )
```

For example, for the function `m1`, which is nested to a depth of 2, variable `ind` will be (2, 0) and `x` will be (2, 1). Such a two-component addressing is used to locate a variable in the stack as follows:

Local variable: Having B-value equal to current block level, N locations down from the beginning of the local area.

Global variable: Its B-value must be less than the current block level, if so, the base of the AR can be found from an appropriate pointer.

For example, if `x` is accessed within Block 4, its two-component addressing would be (1, 2), its B-value is less than 3 (B-value of Block 4) and it can be accessed using BP(1) and $N = 2$.

The rules for setting up the Static links within an AR are given below:

when a Block-b at B-value $B(j)$ is called from a Block-a at B-value $B(i)$:

if $j = i + 1$: i.e. entering a block nested within: copy all the static links for Block-a into AR for Block-b and add BP(a).

if $j \leq i$: i.e. calling a procedure global to this block: the first $j - 1$ static links in the AR of Block-a are inserted as static links in AR for Block-b.

We should also give the rule for Run-Time address calculations, which will be required to generate proper code. Given an address (B, N) of a variable which is accessible within a block at level L, the address A of the variable is found as:

```
A = if B = L then BP + (B - 1) + Nip + N
             else if B < L then static-link(B) + (B - 1) + Nip + N
                           else error.
```

where BP = current AR's BP, static link(B) = static link for block level B and Nip = number of implicit parameters (two for our running example).

Note that many compilers start the assignment of N not from 0 but from the value $(B - 1) + \text{Nip}$, as both these values are known at the compile-time. This reduces the formula for address calculation to BP + N for locals and static link(B) + N for globals.

7.1.4 Nested Functions in GCC Extension

In the recent version of GNU GCC, several useful extensions are provided in the C compiler. Our immediate interest is in "Nested Functions" as given in "Using the GNU Compiler Collection (GCC)". We compiled the following skeleton code representing our running example:

```
int func(){
  float x, y;
  int n;
  //   ...
  void m1(int ind){
    int x;
    //  ...
    m2(ind + 1);
    // ...
  }
```

```
  void m2(int j){
    // ...
    {
      int a[j];
      //  ...
    }
    // ...
  }
  m1(x * y);
}
```

In order to see how gcc handles the variable references, we obtained the assembly language code (without any kind of optimization):

```
.text
.globl func
.type func, @function
func:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
flds -12(%ebp)       # x (float)
fmuls -8(%ebp)       # y (float)
# compute x * y,     convert to int and put in %eax
subl $12, %esp
pushl %eax           # int(x * y) in eax
call m1.1282
addl $16, %esp
leave
ret
.size func, .-func
.type m2.1287, @function
m2.1287:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl %esp, %eax
movl %eax, %edx
movl 8(%ebp), %eax   # j
# block 4 starts here
# compute space for j integers, put next higher multiple of 16 in eax
subl %eax, %esp      # make enough space for a[j]
movl %esp, -20(%ebp)
# block 4 ends: readjust SP for 16-byte alignment
# it might have been disturbed by block 4
movl %edx, %esp
leave
ret
.size m2.1287, .-m2.1287
.type m1.1282, @function
```

```
m1.1282:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl 8(%ebp), %eax
incl %eax             # increment ind
subl $12, %esp
pushl %eax            # call m2(ind + 1)
call m2
addl $16, %esp
leave
ret
.size m1.1282, .-m1.1282
```

As can be seen from a close study of the above assembly code, the compiler itself computes all addresses which are via the static links, and the dynamic links are resolved by usual addressing with respect to `%ebp`.

As a further example, we compiled the following two programs:

```
int main(){
  int i = 5;
  void func(){
    i = 9;
  }
  func();
  printf("%d\n",i);
}
```

This program gave the following output:
9

```
int main(){
  int i = 5;
  void func(){
    int i;
    i = 9;
  }
  func();
  printf("%d\n",i);
}
```

whereas this program gave the output:
5
showing that the locality of the variables declared within the nested function is maintained. The following two listings give the relevant portions of the assembly code output, which explains how the "on the fly" static link works.

```
main:
...   ...   ...   ...
       movl    $5, -12(%ebp)
```

```
        leal    -12(%ebp), %ecx
...    ...    ...    ...
        pushl   %ebp
        movl    %esp, %ebp
        movl    $9, (%ecx)
        popl    %ebp
        ret
```

In the above code, which corresponds to the situation where `func()` refers to variable `i` global to itself, the calling routine puts the *address* of `i` in register `%ecx`. The function `func()` assigns a value 9 to this address, which is the global variable. Thus, register `%ecx` effectively behaves like a static link.

```
main:
...    ...    ...    ...
        movl    $5, -8(%ebp)
...    ...    ...    ...
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $9, -4(%ebp)
        leave
        ret
```

In this case, storing the address of `i` is missing and as such `func()` puts value 9 in its local variable at `-4(%ebp)`.

GCC uses a method called "trampoline", which refers to a technique for implementing pointers to nested functions. The trampoline is a small piece of code which is constructed on the fly on the stack when the address of a nested function is taken. The trampoline sets up the static link pointer, which allows the nested function to access local variables of the enclosing functions. The function pointer is then simply the address of the trampoline. This avoids having to use "fat" function pointers for nested functions which carry both the code address and the static link.

7.1.5 On-demand or Heap Allocation

The second dynamic storage allocation method is on-demand heap allocation. This allocation strategy consists of pre-reserving a large contiguous block of storage, known as the heap, at compile-time (though the actual memory allocation will be generally done by the Loader, see Appendix C). During the execution of a program, a Run-Time storage management routine, called memory allocator, allocates space from this reserved area on demand. The demand may be explicit, for example C library `malloc()` function, or implicit as in some languages like SNOBOL or LISP.

It is expected of the programmer (for explicit demand) or a part of the language run-time system of a language (for implicit demand), to release any such allotted storage which is no more in use. In some languages, this recovery of unused storage is automatically done periodically by an associated function called *Garbage Collector*.

Implicit storage demand occurs in some languages due to semantics of the language and the programmer does not explicitly write a statement requesting the memory. The compiler of the language will detect that need and automatically generate the requests.

Explicit allocation occurs in languages like C, C++, Java, etc. as a result of programmer-specific requests, via certain library functions. The programmer is responsible for requesting and releasing the memory as and when it is required in his program. It is possible that the system is not able to grant his request for the memory amount asked, and then it is the programmers' responsibility to detect that situation and deal with it.

The heap memory allocation is really a function of the operating system; however, we summarize here its main aspects.

Allocation of fixed length blocks: Used in languages like LISP. All the blocks in the heap memory are kept in the form of two huge linked lists – one of blocks already allocated, pointed to by `alloc` and another of free blocks available for allocation, pointed to by `free`. Two functions `get_block()` and `free_block()` and two counters `freeblk` and `totalblk` are used. When `get_block()` is called it first checks that a free block is available and then scans the free list and allocates the first available free block, puts it in the allocated list, removes it from the free list and returns a pointer to it after decrementing `freeblk`. Note that the `next` pointer required for list management is within the block, and the pointer value returned to the requester is $p + 1$ if p is the address of the free block allocated (see Fig. 7.14(a)). There is a small problem with the arrangement shown in Fig. 7.14(a). Immediately alter obtaining a block, a malicious programmer can get the list link value and possibly disturb the remaining list. The block is generally cleared to zeros before it is passed on to the requester. Figure 7.14(b) shows a somewhat safer arrangement.

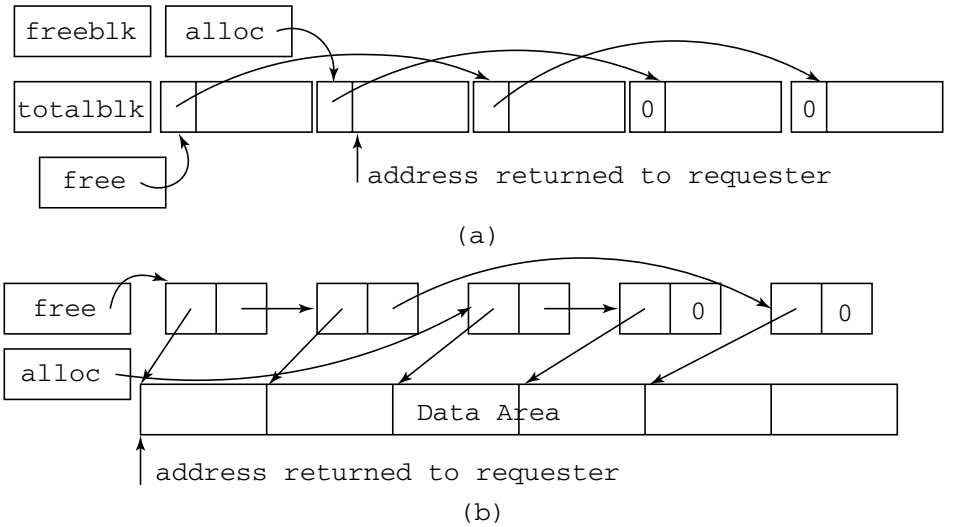


Fig. 7.14 Heap: fixed-sized blocks. (a) Basic concept. (b) A safer implementation

Allocation of variable length blocks: In this scheme, the list of available blocks contains blocks of varying sizes. Each block now contains two administrative fields – pointer to the next block and size of this block. When a block is freed, adjoining free blocks are joined to form a larger free block. There is a problem of *fragmentation*, which manifest itself in two ways:

- **External fragmentation:** A request for allocation is refused because the total space is divided into a large number of small free spaces, enough to grant the request if they were adjoining.
- **Internal fragmentation:** A block much larger than what is requested is the only next larger available and is allocated, wasting much of the allocated space.

There are several algorithms which try to reduce adverse effects of these problems:

- **First fit algorithm:** The free-block list is linked in the order of increasing memory address and a free block larger than the request is allocated. This algorithm allows easy joining of adjoining free blocks and reduces the chance of external fragmentation.
- **Best fit algorithm:** The free blocks are linked in the increasing order of their sizes. The first free block larger than the request is allocated. This algorithm reduces the chance of internal fragmentation.
- **Buddy-system algorithm:** Both the above methods have their problems. In buddy-system, the blocks are of fixed, pre-determined sizes, out of a set of possible values. Usually a Binary buddy-system is used, where the fixed sizes of the blocks are power of 2, 2^n for different values of $n > t$, where t is a threshold for each hardware platform. For example, in GNU GCC compiler, $t = 3$, making minimum block size to be 8-bytes. Each block of size b in available memory has a *buddy* block, of the same size and physically adjoining, either before or after it in memory. The idea is that a block of size b, if free, can be combined with its buddy, if it is also free, to get a free block of size $2b$. With each block a single bit `tag`, in a bit array, is associated. The tag bit is 1 if the block is free and 0 if it is allocated.

The memory manager maintains several lists of free blocks of same size 2^n, for example, the allocator in Linux operating system has lists of blocks of size 32, 64, 128, …, 131072 bytes.

For a request of block-sized n, find a block of size $m \geq n > k$, where $m = 2^p$ and $k = 2^{p-1}$ out of the fixed-sized blocks. If such an m-sized free block is not available, then split repeatedly a larger-sized block, of size say 2^q, $q > p$, to get a block of just correct size. This method also simplifies the joining of free blocks.

Freeing of allocated blocks: There are several ways in which unused blocks can be freed.

- **Free-as-you-go:** There are two methods:
  1. **Explicit freeing:** The programmer has the responsibility of freeing the unused blocks, as in case of C language, by `free()` library function.
  2. **Implicit freeing:** Each block has a reference counter, when it reaches zero, free the block, during some operation in which the block would have been freed.
- **Garbage collection:** This method uses a special daemon routine which checks all unused blocks periodically and returns them to availability list. When is the checking done?
  1. when the availability list has become too small;
  2. a request for memory could not be met;
  3. time-slice driven.

Basically, there are two steps: mark the blocks being referred in any of the programs; reclaim the unmarked blocks and return them to the free list.

Compactation: The free blocks are moved in the memory such that a single large contiguous available block results. However, this method is very time consuming.

Garbage Collector Methods

A garbage collector typically uses a criterion defined in terms of a *root set* and *reachability* from these roots. At the point when garbage collection occurs all globally visible variables of active procedures are considered live and so are the local variables of any active procedures. The root set therefore consists of the global variables, local variables in the activation stack and any registers used by active procedures. Heap objects directly reachable from any of these variables could be accessed by the running program, so they must be preserved. In addition, since the program might traverse pointers from those objects to reach other objects, any object reachable from a live object is also live. Thus, the set of live objects is simply the set of objects on any directed path of pointers from the roots. Any object that is not reachable from the root set is garbage, i.e. useless, because there is no legal sequence of program actions that would allow the program to reach that object. Garbage objects therefore cannot affect the future course of the computation, and their space may be safely reclaimed.

We shall discuss here one of the methods – reference counting.

Reference counting: In a reference counting system, each object has an associated count of the references (pointers) to it. Each time a reference to the object is created, e.g. when a pointer is copied from one place to another by an assignment, the object's count is incremented. When an existing reference to an object is eliminated, the count is decremented (see Fig. 7.15). The memory occupied by an object may be reclaimed when the object's count equals zero, as this indicates that no pointers to the object exist and the running program could not reach it.

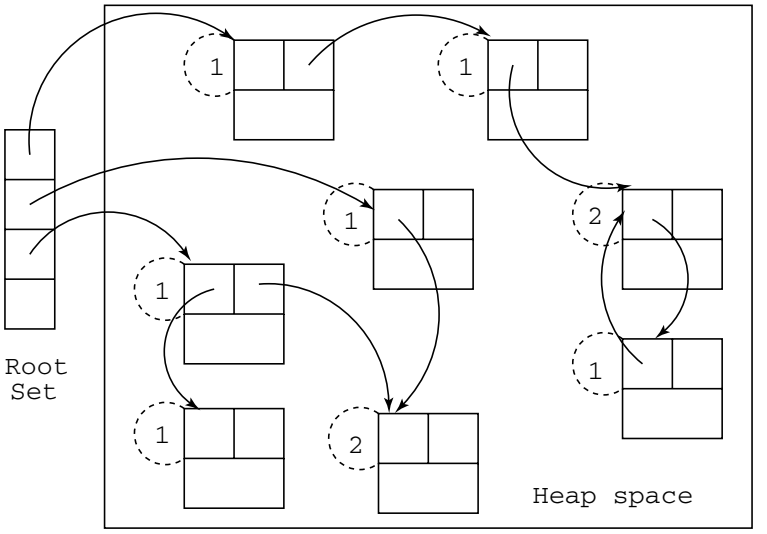


Fig. 7.15 Reference counting. In a straightforward reference counting system, each object typically has a header field of information describing the object, which includes a subfield for the reference count. Like other header information, the reference count is generally not visible at the language level

When the object is reclaimed, its pointer fields are examined, and any objects for which it holds pointers to also have their reference counts decremented, since references from a garbage object do not count in determining liveness.

Reclaiming one object may therefore lead to the transitive decrementing of reference counts and reclaiming many other objects. For example, if the only pointer into some large data structure becomes garbage, all of the reference counts of the objects in that structure typically become zero, and all of the objects are reclaimed.

One advantage of reference counting is the incremental nature of most of its operation – garbage collection work, i.e. updating reference counts, is interleaved closely with the running program's own execution. It can easily be made completely incremental and real time, that is, performing at most a small and bounded amount of work per unit of program execution.

There are two major problems with reference counting garbage collectors; they are difficult to make efficient, and they are not always effective. The effectiveness problem is that reference counting fails to reclaim circularly linked structures. If the pointers in a group of objects create a directed cycle, the objects' reference counts are never reduced to zero, even if there is no path to the objects from the root set. This is the *Problem with Cycles*.

7.1.6 Parameter Passing and Calling Conventions

The protocol by which a subroutine is invoked and it returns the computed value back to the caller routine is termed as *Calling Convention*. There are several calling convections in use, many of them non-compatible. Basically, there are two types:

Caller clean-up: In this type of conventions, the caller cleans the call arguments from the run-time stack, which allows for variable argument lists, e.g. in C library function `printf()`.

The most frequently used convention of this type is *cdecl* calling convention, which is used by many C systems for the x86 architecture. In `cdecl`, function parameters are pushed on the stack in a right-to-left order. Function return values are returned in the EAX register (except for floating-point values, which are returned in the x87 register ST0). Registers EAX, ECX and EDX are available for use in the function. The calling function cleans the stack after the function call returns.

On Linux, GNU C compiler `gcc` sets the *de-facto* standard for calling conventions. From `gcc` 4.x the stack must be aligned on a 16-byte boundary when calling a function, the previous versions only required a 4-byte alignment.

Callee clean-up: In this type of conventions, the called routine has to clean the stack just before returning. When the callee has responsibility of cleaning the call arguments from the stack, it needs to be known at compile-time how many bytes the stack needs to be adjusted. Therefore, these calling conventions are not compatible with variable argument lists, e.g. `printf()`. However, they may be, slightly more space efficient, as the code needed to unwind the stack does not need to be generated by the calling code.

Functions which utilize these conventions are easy to recognize in ASM code because they will unwind the stack prior to returning. The x86 `ret` instruction allows an optional 16-bit parameter that specifies the number of stack bytes to unwind before returning to the caller. Such code looks like this: `ret 12`.

A strategy for passing call arguments to a function or a procedure defines when and in what order the arguments to a function are evaluated, when they are substituted into the function, and what form that substitution takes. Such strategies are divided into two basic groups, *strict* and *non-strict*, based on how arguments to a function are handled. A language may use several argument passing strategies, for example C and C++ use call-by-value and call-by-reference. Many strict languages use some form of non-strict evaluation for Boolean expressions and if-statements.

Strict evaluation of arguments: The arguments to a function are always evaluated completely before the function is evaluated. Most existing programming languages use strict evaluation for functions.

Non-strict evaluation of arguments: The arguments to a function may not be all evaluated completely before the function is evaluated. The arguments are not evaluated unless they are actually used in the evaluation of the function body. This method of evaluation is also called *lazy* evaluation.

Pass-by-Value

Pass-by-value, also called *call-by-value* evaluation, is the most common evaluation strategy, used in many languages like C, C++, Java and Scheme. In pass-by-value, each argument expression is evaluated first, and then the resulting value is bound to the corresponding formal argument of the function. If the function or procedure is able to assign values to its parameters, only its local copy is assigned, anything passed through a function call is unchanged in the caller's scope when the function returns. Some languages like Java evaluate the call arguments from left to right, while some other, like C, evaluate them right to left. The order of argument evaluation becomes a significant factor if the call argument expressions have side effect. For example, consider the following code segment:

```
void func(int x, int y){
     printf("%d %d", x, y);
}
int main(){
     int n = 5;
     func(n, n++);
}
```

As a C language code, this will print "6 5", but an equivalent Java code would print "5 5". If the order of evaluation of call arguments is not specified for a language, different compiler implementors may use different order, and it can result in confusion.

It is possible to pass a reference value or a pointer to a variable, which will not be modified by the called function, but the value of the variable can be changed, as the value which is passed is not the value of the variable as understood by the ordinary meaning of value, but an implementation-specific reference to the value. It is sometimes called `pass-by-sharing`. Thus, one should not take the pass-by-value as a means of information hiding or protection. In C if you give pointer to some item to a function, you are allowing the possibility of that function changing the data item.

Pass-by-Reference

In pass-by-reference method, also called *call-by-reference* evaluation, a function is given a reference or a pointer to a variable as the argument, rather than a copy of its value. This would mean that the function can modify the referred argument variable – and the effect of this modification will be seen by the caller. Call-by-reference therefore has the advantage of greater space efficiency, since arguments do not need to be copied, as well as the possibility of greater communication between a function and its caller. In fact, in C language, which does not have a syntactic class of "procedure", which normally would provide a possibility of zero to several returned values, comparable behaviour can be obtained by use of pass-by-reference arguments.

Pass-by-Sharing

Pass-by-sharing, also called as *call-by-sharing*, is used by languages like Python, Ruby, Scheme, OCaml, etc. However, the terms pass-by-sharing and call-by-sharing are not standardized. Pass-by-sharing

generally means that the referred values are objects rather than primitive types. The semantics of pass-by-sharing differ from pass-by-reference in that assignments to the function arguments within the function are not visible to the caller, unlike pass-by-reference semantics. If a variable was passed, it is not possible to effect an assignment on that variable in the caller's scope. However, since the function has access to the same object as the caller, because no copy is made, mutations to those objects, if the objects are mutable, within the function are visible to the caller, which may appear to differ from pass-by-value semantics. For immutable objects, there is no real difference between pass-by-sharing and pass-by-value.

Pass-by-Copy-Restore

Also called as *call-by-copy-restore*, *call-by-value-result* or *call-by-value-return*. This form of argument passing is a special form of pass-by-reference. This form is relevant to multi-threaded programming. In short, any modifications to the call arguments of this type are not seen by the caller till the function actually exits. Thus if the same argument is being passed to some other functions as pass-by-copy-restore, those functions will see the old value, even though the first function has modified it.

Pass-by-Name

In this method, the arguments to a function are not evaluated before the function is called. Their identifications are substituted directly into the function body and then left to be evaluated whenever the function encounters them. If an argument is not actually used in the function body that argument is never evaluated and if it is used several times, it is re-evaluated each time it appears. This is one of the most complex argument passing methods to implement. Effectively, each argument is as if represented by its own evaluation function reference. In the language FORTH, the variables are passed in some what similar fashion.

Pass-by-Need

This method can be considered a stripped downed version of pass-by-name. The argument is not evaluated till the function actually needs it. When the need for the argument value arises, it is evaluated and its value is saved for all future use within the function.

7.1.7 C Variables

Sometimes there is confusion in new C language programmers because the word "static" is used in two distinct senses:

Storage classes: **Static** – persists throughout execution, e.g. `static int i;`

External – always static;

Automatic – created when the block in which they are declared is executed, on exit discarded; variables declared inside a function (unless made static); allocated on the stack (allows recursion).

Linkage classes: qualifier "static" also refers to this.

Internal linkage – cannot be accessed across source files; all variables within a function and all globals declared "static";

External linkage – can be accessed across source files; all variables outside a function, all function names, not declared "static".

As most of the non-trivial real-life programs in C are multi-source file programs, it is necessary to be clear about the use of this particular keyword.

7.1.8 Block-structured Languages

The concept of blocks is related to the scope rules in a language. Though different languages have different scope rules, two most common rules are:

Declaration before use rule: It requires that an identifier be declared before it is used for any purpose. This allows the Symbol Table being constructed as the program scanning proceeds and references to a particular identifier, and updation of its attributes is possible during later phases of compilation. As declaration also normally includes the type of the identifier, the type checking is possible by the compiler, which reduces the programmer errors to a great extent.

Most closely nested rule: Block structure is quite common among modern programming languages. A *block* is a programming construct which can have its own declarations of identifiers. For example in C, the following are considered blocks:

- source file,
- function definition,
- compound statement, i.e. those enclosed in curly brackets – including those in `if`, `while` and `do` constructs,
- in the loop-control parentheses in a `for` header, the range of declared identifier is throughout the range of the `for` construct,
- `struct` and `union` definitions, with some limitations.

A language is *block structured* if:

- it allows nesting of blocks,
- the scope of declarations in a block is limited to that block and blocks contained within it,
- when blocks are nested, declaration in the most closely nested block applies.

Block-structured languages like PASCAL have special requirements for storage allocation and access. C is only partially block structured, as it does not allow definition of a function within a block – all functions are global and hence visible within a source file, though functions declared `static` are not visible in another source file outside a source file in which it is defined.

Basically, the Symbol Table for a non-block structured or partially block-structured language need to have the following two operations:

Operation	Explicit declaration	Implicit declaration
Insert	Declaration	First occurrence
Look-up	Every reference	Every subsequent reference

For block-structured languages like PASCAL, ADA or Delphi, two additional operations are needed – *block begin* and *block end*. Within a nested block, all the variables in the outer block are valid and retain their attributes, unless the same identifier is redefined within the block.

The Symbol Table for such languages are more sophisticated, as we shall now see. Consider a program shown in Fig. 7.16, written in "block-structured C". It is not a real C program, note the functions `m1()` and `m2()` defined within the outer block. As identifiers `m1()` and `m2()` get defined only when the outer block is executed, it cannot be put in the Symbol Table as an active symbol at compile time. Thus, while processing `call m2()` within `m1()`, the Scanner should tag `m2()` as an undefined function, indicated by P? You can see that though the Scanner sees all the identifiers,

some of them are inactive and become active and inactive as the *execution* of the blocks proceed. For example, `x` is a `float` in the beginning of block 1, but when `m1()` gets defined, it becomes `int`. The catch is that the compiler has to mimic the execution-time definitions, redefinitions and removal of identifiers, while processing the program sequentially from beginning to end.

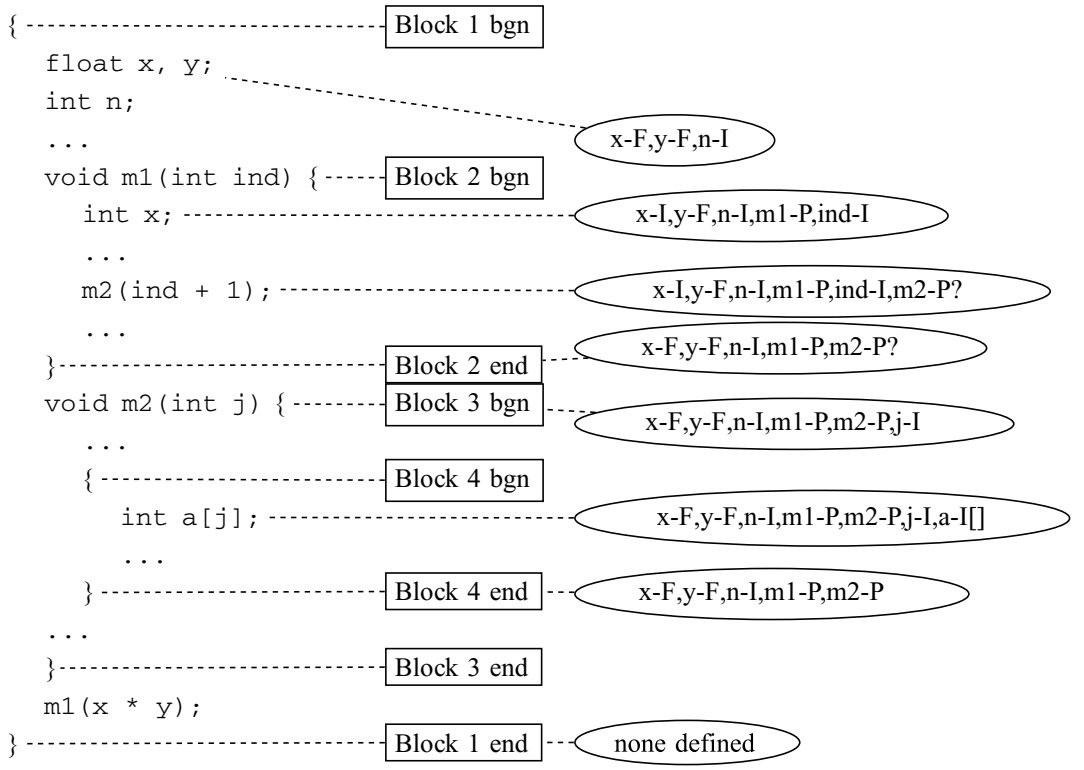


Fig. 7.16 A "block-structured C" program. The left-hand side is the program code, the middle column shows begin and end of blocks and the right-hand side column containing ovals indicate the state of each identifier. Note that the block structure is parenthetical, i.e. it is lexically well structured

The block structure is a properly nested structure and entry and exit from the blocks are like "push" and "pop" of a stack. Also, as in the case of the identifier `x`, the latest definition of an identifier is considered active. This tells us that we should use a stack to manage the Symbol Table for a block-structured language. For building a Symbol Table for such a language, one of the following methods are generally used.

Stack Symbol Table

The Symbol Table is organized in the form of a stack. As identifier declarations are encountered, the corresponding entries are pushed onto the stack. At the end of the block, the corresponding entries are effectively deactivated by reference to an auxiliary *block pointer stack*. Referring to Fig. 7.17, the block pointer stack points to the bottom (beginning) of the stack allocated to a particular block.

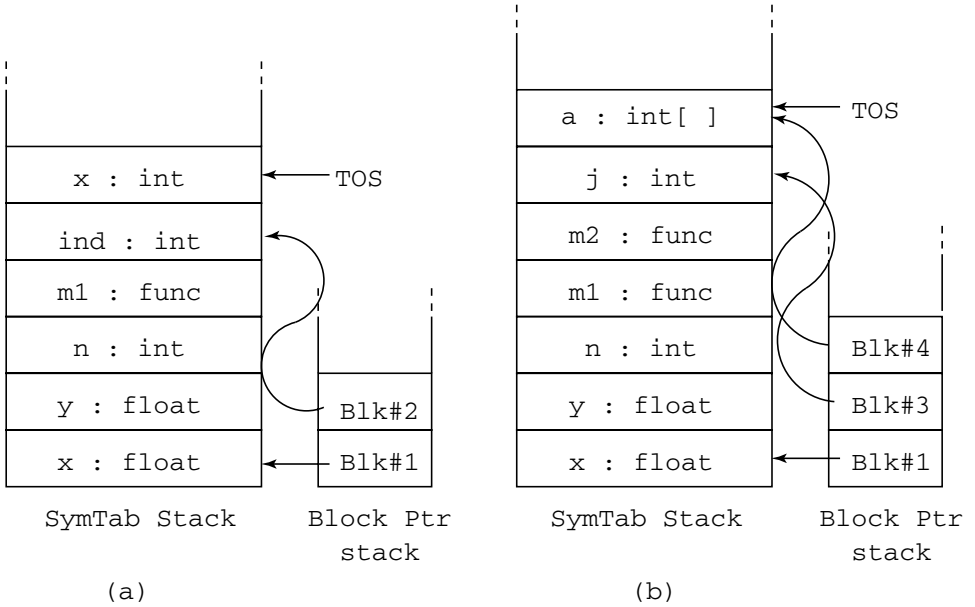


Fig. 7.17 Stack Symbol Table operation. (a) Position just before the end of Block 2. (b) Position just before the end of Block 4

Within a particular block, the identifiers from TOS to the bottom of the stack are active. For example, within block 2, `x, y, n, m1, ind` are active and the first (top-most on the stack) definition of the repeated identifier `x` will be valid. When Block 2 ends, its identifiers `x`, `ind` are popped up from the Symbol Table, the Block 2 pointer is popped up from the auxiliary stack and the identifiers active within Block 1 are available, from TOS downwards. The identifiers active only in Block 2 are no longer active and are not available.

We have seen in the previous chapters that the Symbol Table is referred to and updated with new information as the compilation phases proceed. Also, at the end of compilation we may like to have a list of identifiers for record purpose or for later symbolic debugging. Thus, it is desirable that the identifiers are not irrevocably discarded on the exit from a block.

One possible solution is to have an additional array, having entries with two fields – block begin pointer and block end pointer. The block pointer stack now points to this array, instead of the main stack. Go on stacking the identifiers as they are encountered in the source code. Make appropriate entries in the begin–end array for each block. Within a particular active block, all the identifiers in its range as per the begin–end array are active and so are the identifiers in blocks below it in the block pointer stack, except those repeated (see Fig. 7.18). Actually, we need not have the additional "begin–end" array; the *static links* part of the Activation Record (AR) provides just this information. That would of course mean that during various phases of compilation a stack with the replica of ARs being pushed when a block starts and popped when a block ends has to be used.

Stack-based Binary Search Tree

A Binary Search Tree (BST) as a Symbol Table has obvious advantages compared to a linear list structure, especially for programs having a large population of identifiers. If the BST can be kept

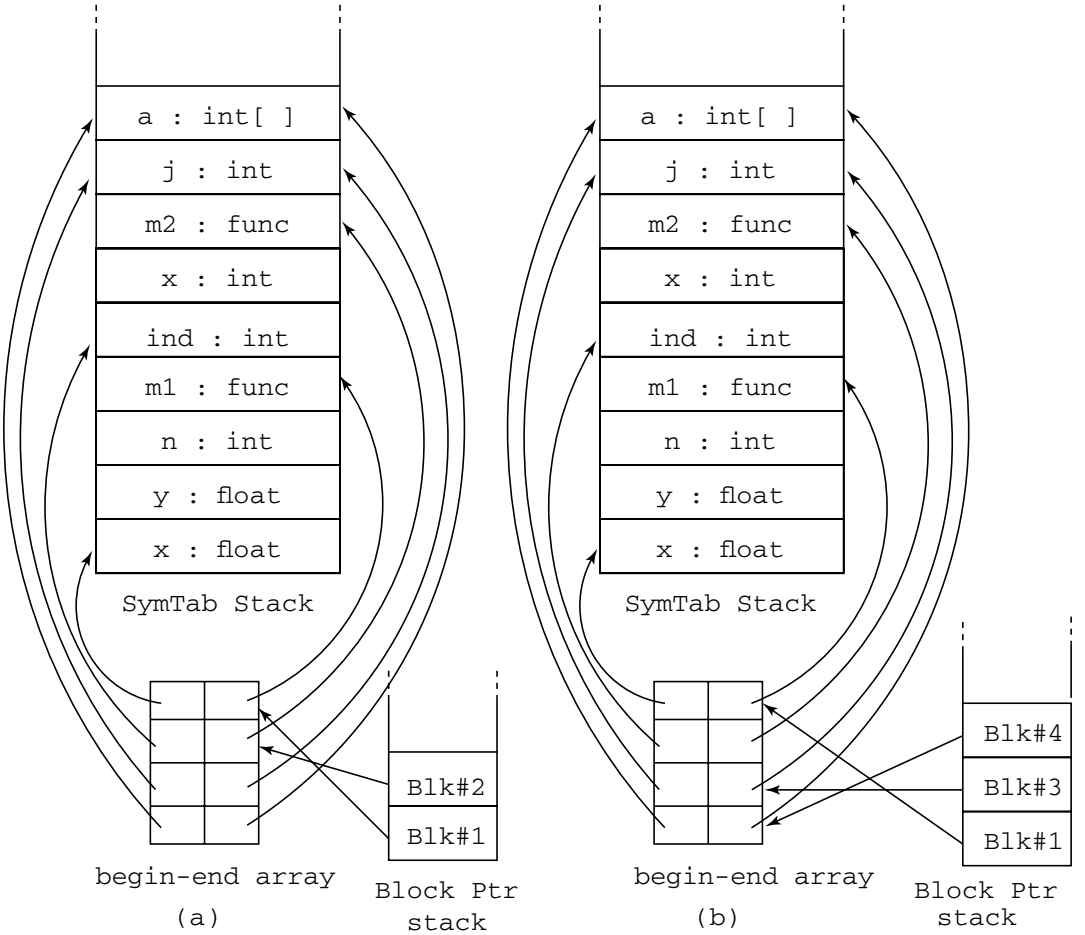


Fig. 7.18 Preserving the discarded identifiers. (a) Position just before the end of Block 2. (b) Position just before the end of Block 4

even approximately balanced as it grows, the identifier search time is reduced considerably due to the expected $O(\log_2 n)$ running time complexity, compared to $O(n)$ time for a linear list. This means that for a Symbol Table with 1000 symbols the time ratio will be approximately 100.

One drawback of BST is the complex and time-consuming delete operation. For a block-structured language, though delete operation is necessary if we consider only parsing of the source code, we may not like to really remove a node from the tree for deactivating an identifier. We will be required to use the Symbol Table in the later phases. Instead, a flag may be used to indicate the deactivated identifier so that the symbol lookup function ignores it. This realization makes the case for the BST-based Symbol Table stronger.

We have already seen some preliminary details about a Symbol Table implementation using BST in Chapters 3–6. Now we show how we can use a stack to control the access to the BST-based Symbol Table, so that the requirements of a block-structured language are satisfied.

We present now a possible approach to BST-based Symbol Table for a block-structured language. First, we define a link field in our Symbol Table node data structure, Node:

```
typedef struct node_struct{
struct node_struct *link[2];
util u,v,w,x,y,z;
}Node;
#define L(n) ((n)->link[0])       // symTab BST admin
#define R(n) ((n)->link[1])       // symTab BST admin
#define name(n)     ((n)->w.S)    // Symbol Table data
#define type(n)     ((n)->v.I)    // Symbol Table data
#define ival(n)     ((n)->u.I)    // Semantic analysis
#define ttree(n)    ((n)->z.N)    // type checking constructor
#define hocsym(n)   (Symbol *)((n)->y.P) // interpreter
#define hocinst(n)  (Inst *)((n)->y.P)
#define hocnarg(n)  ((n)->y.I)
#define subtype(n)  ((n)->y.I)    // type checking - subtype
#define nxtinblk(n) ((n)->x.N)    // ptr to node next in block
```

Thus, the field `nxtinblk(n)` is a pointer to the next symbol node in the Symbol Table, which is the next identifier in this block. Apart from defining this link in the node, we have to use a *block* stack made up of an array of Node pointers:

```
int ibstk = -1;
Node * bstack[1024];
bpush(Node * b){
  if(ibstk < 1023) bstack[++ibstk] = b;
  else fprintf(stderr,"bstack overflow\n");
}
Node * bpop(){
  if(ibstk >= 0) return bstack[ibstk--];
  else fprintf(stderr,"bstack underflow\n");
}
```

We build the Symbol Table normally during the Scanner phase, by inserting the identifiers as they are encountered. When a block beginning is detected we create a fresh Node having the block number as data, put it in the Symbol Table as if it were an identifier and push a pointer to it on the block stack. This node will act as a header for the linked list that we are going to build up as new identifiers are detected in this block.

```
int blockno = 0;
Symbol * insertblock(){
        char blkno[12];
        Symbol *s;
        sprintf(blkno, "blkbgn%d", blockno);
        s = install(blkno,VAR, blockno++);
        return s;
}
```

When an identifier is encountered, first the BST management routine `insert()` will put a corresponding Node in the Symbol Table and then it will be linked to the header node in the block stack.

```
void blklink(Symbol *blknoptr, Symbol *np){
        Symbol *p;
        if(blknoptr != NULL){
                p = nxtinblk(blknoptr);
                nxtinblk(blknoptr) = np;
                nxtinblk(np) = p;
        }
}
decl:    dtype ':' dlist
        ;
dtype:   IVAR { }
        | VAR { }
        | SVAR
        ;
dlist:   VAR            {type($1) = type($<sym>0); blklink(blknoptr,$1);}
        | dlist ',' VAR  {type($3) = type($<sym>0); blklink(blknoptr,$3);}
        ;
```

On exit from a block, its header is popped off from block stack.

Search for a match of an identifier (lookup) is done from top of the block stack downwards, first looking through the linked list at the current block level, then in the linked list of the next enclosing block, etc.

We implemented such stack-based BST table as described above and tested it with the following code, in a stripped down C-like language:

```
int : a,c        // block 0
{
int : b,d        // block 1
a = 3
{
int : e,f        // block 2
b = 5
{
int : g,h        // block 3
c = 7
}
d = a * (b + c)  // block 2
}                // block 1
}                // block 0
end
```

The outermost block is numbered 0, and the inner blocks are sequentially numbered 1, 2, etc. The Symbol Table access is indicated by the following trace:

```
[blkbgn3]
[h](275){0}
[g](275){0}
```

```
[blkbgn2]
[f](275){0}
[e](275){0}
[blkbgn1]
[d](275){0}
[b](275){0}
[blkbgn0]
[c](275){0}
[a](275){0}
```

In the above trace of the block pointer stack, the identifiers linked in each block number are printed under the block number. The integer value 275 is the internal Scanner code for INT variable. The third field, zero for all variables, indicates current value, which, as we have not actually executed the program, are all zeros.

Stack-based Hash Table

We can also use hash table along with a control stack as a Symbol Table for block-structured languages. In fact, if we decide to use chaining as a means of dealing with collision in the hash table, then the resulting hash table consists of an array of pointers to linked lists, each corresponding to a hash value. We can use, instead of hashing, the current block number as an index to the pointer array and put all the identifiers encountered in the current block in the associated linked list (see Fig. 7.19). The active or non-active identifiers are decided by means of a block pointer stack, similar in principle to the stack-based BST implementation discussed above. This arrangement should work well for small-to-medium sized source programs, with block nesting of reasonable depth.

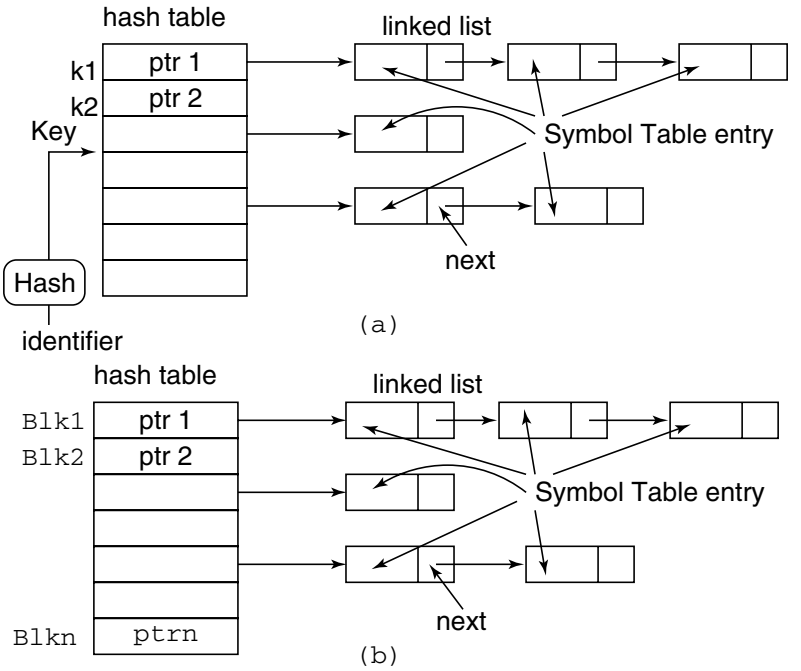


Fig. 7.19 Using an entry in the pointer array to represent a block

7.2 Operating System

The operating system manages resources – CPU time and Memory – and allocates them to requesting programs. Very roughly, a running program is represented by a *Process* within the O/S. We shall have a brief review of how an O/S manages the processes.

All the system services are accessed via System Call Interface. We shall see some examples of assembly language programs using a few common system calls.

7.2.1 A Running Program – A Process

On a Unix-like system, including Linux, many user and system processes run concurrently (are active simultaneously). The one the user has invoked usually runs in the foreground, the other processes run in the background. The command `ps` will show you which processes are active and what PID numbers are assigned to these processes. The command `kill` allows the user to get rid of any process. Without option this is a request: "please go away", but `kill -9` followed by the process PID is an immediate kill. Foreground processes can often be killed by typing `Control-C`.

A process is represented within the O/S kernel by a data structure called process control block (PCB), which contains all the administrative data about a process, like allocated memory areas, space for saved CPU registers, process state, etc.

The operation of the multi-tasking executive is almost completely transparent to a normal programmer and is seldom his concern. The only significant fact which emerges out of the working of the O/S, as far as a compiler writer is concerned, is the availability of the virtual memory and its protection system. In a typical O/S, for example in Linux, a flat memory of rather large size (4 Gbytes in Linux) is seen by the programmer, out of which 1 Gbyte are reserved for the Kernel use, for providing various services to the program.

If a programmer has the responsibility of developing the language library, then he will have to be familiar with other aspects of O/S.

7.2.2 Linux System Calls

On Unix, Unix-like and other POSIX-compatible operating systems, popular system calls are open, read, write, close, wait, execve, fork, exit and kill. Many of today's operating systems have hundreds of system calls. For example, Linux has over 300 different calls, and FreeBSD has almost 500.

The C library uses these System Calls for accessing various system services. For example, C library `read()` is just a wrapper for `sys_read` = 3.

Example 1: Simplest program using System Calls

Here is a very simple assembly language program, using a system call. It is in GNU assembler format.

```
# Text segment begins
.section .text
   .globl _start
# Program entry point
    _start:
# Put the code number for system call
          movl $1, %eax
/* Return value to environment*/
```

```
          movl $2, %ebx
# Call the OS
          int $0x80
```

The above program does not have a `.data` segment. System Call number 1, i.e. SysCall-1, denoted by putting the value 1 in register EAX, terminates a program and returns control back to the O/S. It also returns a *return code* to the O/S environment by setting its value in register EBX. This value is available as value of the shell variable `$?` and can be read by executing the command `echo $?`. This little program does not do anything significant except that it returns a small integer (2) and terminates.

Note that in the GNU assembler source code format, the general form of an instruction is `OP SRC, DST,` which is reverse of the Intel assembly language format `OP DST, SRC`. Constants are prefixed by $ and register names by %. Hex constants are written starting with 0x. The operand size is indicated by a suffix to the op-code. Thus, `movl` denotes `move` long (4-bytes), `movw` moves 2-bytes and `movb` moves 1-byte.

Our second example uses three system calls – one for `sys_read = 3`, one for `sys_write = 4` and one for `sys_exit = 1`. It gives a prompt for some text entry, reads it and then writes a message. The program also illustrates the use of *macros* to reduce the program writing effort.

Example 2: Read/Write using System Calls

```
.section .data
   prompt_str:
       .ascii "Enter Your Name: "
   pstr_end:
       .set STR_SIZE, pstr_end - prompt_str
   greet_str:
       .ascii "Hello "
   gstr_end:
       .set GSTR_SIZE, gstr_end - greet_str
.section .bss
// Reserve 32-bytes of memory
   .lcomm buff, 32
// Macros:
// implements the write system call
   .macro write str, str_size
       movl $4, %eax        // 4 = sys_write
       movl $1, %ebx        // 1 = STDOUT
       movl \str, %ecx
       movl \str_size, %edx
       int   $0x80
   .endm
// Implements the read system call
   .macro read buff, buff_size
       movl $3, %eax        // 3 = sys_read
       movl $0, %ebx        // 0 = STDIN
       movl \buff, %ecx
       movl \buff_size, %edx
       int   $0x80
   . endm
.section .text
```

```
    .globl _start
    _start:
        write $prompt_str, $STR_SIZE
        read $buff, $32
// Read returns the length in eax
        pushl %eax
// Print the hello text
        write $greet_str, $GSTR_SIZE
        popl  %edx
// edx = length returned by read
    write $buff, %edx
    _exit:
        movl $1, %eax
        movl $0, %ebx
        int  $0x80
```

The program was assembled, linked and executed as follows:

```
as -o test2asm.o test2asm.S
ld -o test2asm test2asm.o
./test2asm
Enter Your Name: Pravin
Hello Pravin
```

The System Call Interface using the INT instruction, where the system call number was placed in the EAX register before interrupt 0x80 was executed, is a slightly dated technique. For CISC architectures such as later x86 series processors additional techniques are available. For example, we have newer interface in SYSCALL (SYSENTER), SYSRET (SYSEXIT) (the two mechanisms were independently created by AMD and Intel, respectively, but in essence do the same thing). These are *fast* control transfer instructions that are designed to quickly transfer control to the O/S for a system call without the overhead of an interrupt. Linux Kernel 2.5 began using this on the x86 processors.

System calls can be roughly grouped into five major categories:

1. Process control,
2. File management,
3. Device management,
4. Information maintenance,
5. Communication.

Table 7.1 gives a few of the more frequently used Linux System Calls for Linux Kernel version 2.4.x.

While giving a system call the CPU registers are used to pass the arguments. The function number is put in CPU register `%eax`, the first argument in `%ebx`, ..., third in `%edx`, fourth in `%esi` and fifth in `%edi`. In a rare case where more than five arguments are to be passed, the additional ones are pushed on the stack. Documentation of individual SysCall should be consulted to get details of the arguments to be passed.

For further information, see the Linux Kernel source directory

`/usr/src/linux/include/asm-*/unistd.h,`

where the wildcard * denotes the machine architecture, e.g. `i386`, `x86_64`, `arm`, etc.

Table 7.1 Some of frequently used Linux System Calls

No.	Name	Description	Source
1	exit	Terminate the current process	kernel/exit.c
2	fork	Create a child process	arch/i386/kernel/process.c
3	read	Read from a file descriptor	fs/read_write.c
4	write	Write to a file descriptor	fs/read_write.c
5	open	Open a file or device	fs/open.c
6	close	Close a file descriptor	fs/open.c
7	waitpid	Wait for process termination	kernel/exit.c
8	creat	Create a file or device	fs/open.c
9	link	Make a new name for a file	fs/namei.c
10	unlink	Delete a name and possibly the file it refers to	fs/namei.c
11	execve	Execute program	arch/i386/kernel/process.c
12	chdir	Change working directory	fs/open.c
13	time	Get time in seconds	kernel/time.c
14	mknod	Create a special or ordinary file	fs/namei.c
15	chmod	Change permissions of a file	fs/open.c
16	stat	Get file status	fs/stat.c
17	lseek	Reposition read/write file offset	fs/read_write.c
18	getpid	Get process identification	kernel/sched.c
19	mount	Mount filesystems	fs/super.c
20	umount	Unmount filesystems	fs/super.c
21	setuid	Set real user ID	kernel/sys.c
22	getuid	Get real user ID	kernel/sched.c
23	utime	Set file access and modification times	fs/open.c
24	access	Check user's permissions for a file	fs/open.c
25	nice	Change process priority	kernel/sched.c
26	sync	Update the super block	fs/buffer.c
27	kill	Send signal to a process	kernel/signal.c
28	rename	Change the name or location of a file	fs/namei.c
29	mkdir	Create a directory	fs/namei.c
30	rmdir	Remove a directory	fs/namei.c
31	dup	Duplicate an open file descriptor	fs/fcntl.c
32	pipe	Create an interprocess channel	arch/i386/kernel/sys_i386.c
33	times	Get process times	kernel/sys.c

Tracing the Linux System Calls

The system calls issued by an executable program can be traced on Linux and Unix-like systems by the utility `strace(1)`. In the simplest case, strace runs the specified command until it exits. It intercepts and records the system calls which are called by a process and the signals which are received by a process. The name of each system call, its arguments and its return value are printed on standard error or to the file specified with the -o option.

It is a useful diagnostic, instructional and debugging tool. It is invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. Students can learn a great deal about a system and its system calls by tracing even ordinary programs.

For example, we used `strace` with Example 2 and obtained the following output:

```
$ strace ./test2asm
execve("./test2asm", ["./test2asm"], [/* 67 vars */]) = 0
write(1, "Enter Your Name: ", 17Enter Your Name: )       = 17
read(0, Pravin
"Pravin\n", 32)               = 7
write(1, "Hello ", 6Hello )                     = 6
write(1, "Pravin\n", 7Pravin
)                  = 7
_exit(0)                                = ?
Process 19744 detached
$
```

Note that `strace` waited for user input after `read(0`, and the user typed in `Pravin`.

7.3 Libraries

A library is basically just a searchable archive of object files. If you have a number of source files that contain just functions and related data, you can turn these source files into libraries. Libraries are linked with the translated user program in a separate *linking* phase before the program becomes executable. Libraries can be defined to be linked statically or dynamically by a program. A library may be shared between concurrently running programs, then we have a shared library. Thus, we have three kinds of libraries – static, shared and dynamic. Each of these three types has its own advantages and disadvantages. Use of libraries to organize your total application development is generally desirable because it is good for program modularity, and code reuse – *Write Once*, *Use Many times*.

The following are the three kinds of libraries:

Static: Such a library is linked with the executable at compile-time and gets integrated with it. This results in the executable being much larger in size, but the advantage is that the final program is less dependent on versions of libraries.

Shared: Such a library is linked with the load-module at load-time, although needs to be included in the compilation command, so that compiler can generate proper address linkage information. The library does not get integrated with the executable. This results in a much smaller executable size. The down side is that the final program becomes highly dependent on versions of libraries.

Dynamic: Such a library is linked with a running program, i.e. at execution time. In C language library, there are special `dlxxx()` functions to do actions required for linking in and unlinking the

library. The programmer has responsibility of linking the correct library. The library itself can be static or shared. The final program text size can be kept small by this method.

See Appendix C for further details about the linking and loading processes.

7.3.1 Language Library

Each High Level language generally requires the use of a run-time library of functions which are not fully implemented in the CPU hardware. Effectively, such a library extends the capability of the hardware platform and provides a more comfortable operating environment from the viewpoint of the language syntax and semantics. For example, it is quite common to have multi-dimensioned arrays as a data structure in many languages. The basic CPU does not have a direct means of providing the required array element addressability. It is the language library which provides routines to define, manipulate and provide means of element access of such arrays. Generally, due to the close co-ordination required, the language compiler developer group will prepare the language library. For C language, the library is known as `libc`, though there are many versions of it.

Most of the High Level languages also provide basic mathematics operations in the form of library functions, which is provided via what is known as `math` library. While in the language library the emphasis is on recreating the correct semantics of the language, the maths library developer is more concerned with mathematical correctness of the functions, error limits and efficiency.

7.3.2 Special Purpose Libraries

Apart from the basic language and math libraries, a particular application may require special purpose libraries. Some of the more common are – graph and tree functions, numerical methods – like matrices, statistics, special functions – graphics, sound and multimedia operations, etc.

During an application development, a programmer may find it advantageous to create his own library of the functions he finds himself to be using quite often. For example, while preparing examples for this book, the authors found it useful to have a library of binary search tree functions, called `tree-base`. We now summarize the steps necessary to create such libraries.

Creating Libraries: Static Library

The steps are:

1. Create your C source files containing any functions that will be used, the library can contain multiple object files.
2. Compile the files into object files, for example using `-c` option. The object files will have extension `.o`.
3. Give the following commands to create the library:

   ```
   ar rc libmylib.a objf1.o objf2.o objf3.o
   ```

   This will create a static library called `libmylib.a`.
4. Next give the following command:

   ```
   ranlib libmylib.a
   ```

This creates an index inside the library, which allows a fast search for the functions within the library while linking.

Static library usage: Remember to prototype your library function calls in a header file so that you do not get implicit declaration errors. When linking your program to the libraries, make sure you specify where the library can be found, for example,

```
gcc -o foo -L. -lmylib foo.o
```

The `-L.` option tells `gcc` to look in the current directory in addition to the other library directories for finding `libmylib.a`. You can easily integrate this into your Makefile, including even the static library setup part.

Creating Libraries: Shared Library

Creating shared or dynamic libraries is also quite simple. First, while creating the object files, you should use Position Independent Code option:

```
gcc -fPIC -c objf1.c
gcc -fPIC -c objf2.c
gcc -fPIC -c objf3.c
gcc -shared -o libmylib.so objf1.o objf2.o objf3.o
```

The `-fPIC` option commands the compiler to create a Position Independent Code, i.e. create libraries using relative addresses rather than absolute addresses because these libraries can be loaded multiple times at different virtual addresses.

The `-shared` option specifies that an architecture-dependent shared library is being created. Note that we use `gcc` itself to create the library and not `ar` (the archiver) as in the case of the static library.

Shared library usage: To compile the application program using the shared libraries:

```
gcc -o foo -L. -lmylib foo.o
```

Note that it is exactly the same as compiling with a static library. Although it is compiled in the same way, none of the actual library code is inserted into the executable, because it is a dynamic or shared library. You can automate this process using Makefiles.

Since programs that use static libraries already have the library compiled into the program, it can run on its own.

Shared libraries dynamically access libraries at run-time, and thus the program needs to know where the shared library is stored. The actual details of arranging that are somewhat platform dependent and hence not given here.

7.4 System Environmental Variables

In a typical operating system, several system-wide parameters are defined which tune the behaviour of the programs executed on that system. These environment variables, as they are called, can be accessed via the third argument in the `main(int argc, char *argv[], char *env[])` function header. Some of the more common environment variables are:

HOSTNAME name of the computer

BASH the shell program in use

CLASSPATH a list of paths for Java classes and `jar` libraries

HOME the home directory of the current user

HZ the rate of basic timer interrupt

JAVA_HOME place where the Java system is stored

MANPATH a list of paths where manual pages can be found
PS1, PS2 the primary and secondary prompt strings
TERM the type of the Terminal in use
LD_LIBRARY_PATH a list of paths at which various shared libraries can be found
USER account name of the current user
UID identification number of the current user

As you can see, a variety of information is kept in the environment variables.

7.5 Invocation Command-line Parameters

When a command issued to the system shell invokes a program, including a user's program, a number of parameters can be passed to that program from the command-line. These command-line parameters or command-line arguments are accessible within the user program as follows:

C program: In the header of the `main(int argc, char *argv[]), argc` gets from the shell the number of command-line arguments, which include the invocation name of the program as one of them. Thus, the value of `argc` is one plus the number of actual arguments. The arguments are accessible simply as argv[1], argv[2], etc. Further, it is possible that the arguments may be a mix of program *options* and other arguments. To easily handle such a situation, a library function `getopt` is available which help parse the command-line arguments and separate out the options.

Java programs: The command-line arguments are available via the `main (String[] args)` header.

Perl programs: In a Perl program, the command-line arguments are available as $ARGV[0], $ARGV[1], etc. Unlike a C program, $ARGV[0] is the first actual command-line argument.

Assembly language program: The shell passes the command-line arguments to an assembly program just like it does for a C, Java or Perl program. However, the assembly program has to obtain the arguments a bit more "painfully". We give here a sample assembly program to illustrate this:

```
.section .data
// Command table to store upto 10 command-line arguments
   cmd_tbl:
     .rept 10
         .long 0
     .endr
.section .text
   .globl _start
   _start:
// Set up the stack frame
     movl %esp, %ebp
// Top-of-stack contains the number of command-line arguments.
// The default value is 1
     movl (%ebp), %ecx
// Exit if arguments are more than 10
     cmpl $10, %ecx
     jg      _exit
     movl $1, %esi
     movl $0, %edi
// Store the command-line arguments in the command table
```

```
   store_loop:
     movl (%ebp, %esi, 4), %eax
     movl %eax, cmd_tbl( , %edi, 4)
     incl %esi
     incl %edi
     loop store_loop // decr ecx, Loop if Not Zero
// access the command-line arguments
   _exit:
     movl $1, %eax
     movl $0, %ebx
     int  $0x80
```

? EXERCISES

1. Give a layout of the Activation Record for the legacy C compiler function calling conventions discussed in Section 7.1.2. Compare it with the theoretical Activation Record layout. Is there a *static links* area in this AR? Why? Is it possible to access identifier in the outer block? If yes, how can that be arranged?
2. Write a C program divided in two source files to demonstrate the effect of various scope and visibility rules of identifiers in C.
3. Original BASIC language had only 26 variables named *A* to *Z*. How would have this limitation simplified the Symbol Table and Run-Time environment?
4. A student wrote the following code to print values of elements of an array *a*:

```
int a[100];
int main(){
    int n;
    // compute the array a[]
    n = 0;
    while(n < 100)
      printf("a[%d] = %d\n", n, a[n++]);
}
```

   What is wrong with this code, if anything?
5. In C, if a large-sized data structure like an array or a `struct` is to be passed as an argument to a function, generally a pointer to that item is passed. This opens up a possibility for the called function inadvertently modifying the values within the argument data structure. Is it possible to protect against such occurrence? How can you prevent this?
6. Laboratory exercise: Write simple C programs containing individually IF-THEN, IF-THEN-ELSE, WHILE-DO, DO-WHILE, SWITCH-CASE statements, `struct` definitions and array manipulation operations. Compile them to obtain the assembly language output. Prepare a note on how each control construct and data aggregation get translated. [Hint: Use `gcc -c -S -o myprog.S myprog.c` to get assembly output.]
7. (a) Describe the basic structure of an Activation Record (AR).
   (b) Where are ARs stored at run-time? Why?
   (c) Why do you need an AR? What features of a language requires explicit AR and why?
   (d) What are the purposes of the various access links? What language mechanism requires their use? Why?

WEB RESOURCES

1. Richard Jones GC page:
   `http://www.cs.kent.ac.uk/people/staff/rej/gc.html`
2. "A garbage collector for C and C++" Hans-J. Boehm:
   `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`
3. GNU C Library:
   `http://www.gnu.org/s/libc/`
4. GNU Assembler and other BinUtils:
   `http://www.gnu.org/software/binutils`

FURTHER READING

1. **Garbage collection:** Aho, Lam, Sethi and Ullman, "Compilers – Principles, Techniques and Tools", 2nd Ed., Section 7.4.
2. **Buddy-system:** D. M. Dhamdhere, "System Programming and Operating Systems", 2nd Ed., Section 15.1.
3. **Run-Time environment:** Kenneth C. Louden, "Compiler Construction – Principles and Practice", Chapter 7.
4. **Memory allocator:** Kernighan and Ritchie, "C Programming Language", Section 8.7.

GLOSSARY

Addressability refers to the ability of any device or program to access some random access memory.
Aggregation Collection if things.
Aligned is the way data are arranged and accessed in a computer memory. A modern computer reads from or writes to a memory address in word-sized chunks (typically 4-bytes). If the data are stored at a memory offset equal to some multiple of this word size, it increases the system's performance due to the nature of CPU–memory interface. In case of cached memories, 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.
Cdecl calling convention is used by many C systems for the x86 architecture. Function parameters are pushed on the stack in a right-to-left order. Function return values are returned in the EAX register (except for floating-point values, which are returned in the x87 register ST0). Registers EAX, ECX and EDX are available for use in the function. On Linux, gcc sets the *de-facto* standard for calling conventions. This is usually the default calling convention for x86 C compilers, although many compilers provide options to automatically change the calling conventions used.
Diagnostic Diagnosis is the identification of the nature and cause of anything. Diagnostic program is a program written for the purpose of examining the state, or locating problems with the hardware or Operating System.
Immutable Not changeable.
Lazy Executed as late as logically valid.
Permissions Access rights.
Prototype A representative form or pattern.

Reachability It refers to property of being able to go from one vertex in a directed graph to some other vertex. In case of undirected graphs it is sufficient simply to find the connected components in the graph.
Run-time is the period of time during execution of a program and/or environment provided during that time.
Searchable Information is organized so that it can be searched by some query.
Setuid (short for "set user ID upon execution") are Unix access rights flags that allow users to run an executable with the permissions of the executable's owner. They are often used to allow users on a computer system to run programs with privileges set temporarily higher in order to perform a specific task.
Streaming media is multimedia that is constantly received by and presented to an end-user while being delivered by a streaming provider.
Supercomputer is a computer that is at the fore front of state-of-the-art processing capacity, particularly speed of calculation. The main difference between a supercomputer and a mainframe is that a supercomputer employs all its power into executing a few programs as fast as possible, whereas a mainframe uses its power to execute many programs concurrently.
Time-slice is a small duration of time (typically 10 millisec) allowed to processes being executed in round-Robbins fashion in a multi-tasking operating system.
Trampoline is a device consisting of a piece of taut, strong fabric stretched over a steel frame using many coiled springs. People bounce on trampolines. Here the meaning is generally associated with code jumps. In the GCC compiler, trampoline refers to a technique for implementing pointers to nested functions. The trampoline is a small piece of code which is constructed on the fly on the stack when the address of a nested function is taken. The trampoline sets up the static link pointer, which allows the nested function to access local variables of the enclosing functions. The function pointer is then simply the address of the trampoline.
Visualization A presentation to the mind in the form of an idea or image.

8
Intermediate Code

What you will learn in this chapter

- What is Intermediate Code and why do we need it?
- What are different commonly used forms of Intermediate Code?
- How to build a Parse tree in memory?
- What are Polish and Reverse Polish Notation (RPN)?
- How to generate RPN from a parse tree?
- What is N-tuple notation?
- How to generate N-tuples?
- What is an Abstract Syntax Tree (AST)?
- How to generate an AST for a given source code?
- What are Virtual Machines?
- An introduction to PASCAL P-code
- An introduction to Java Bytecode
- What are Threaded codes and what are their variants?
- Implementation of Threaded code interpreter on Motorola 68000 and Intel x86 series machines
- Introduction to SECD and WAM virtual machines
- Real-life: Intermediate codes used in GNU GCC

Key Words

Intermediate codes, reverse Polish notation (RPN), N-tuple notation, abstract syntax tree (AST), virtual machines, P-code, bytecode, threaded codes, SECD, WAM, GNU GCC

The translation of a High Level language program into its machine executable version is performed via a sequence of transformations, which we can roughly divide into two groups – analysis and synthesis steps. Each transformation outputs the program representation in a particular format or language. Especially, the boundary between the analysis and the synthesis parts is usually formed by representation in the form of an *Intermediate* language (see Fig. 8.1).

The Intermediate language form of the program is useful in many ways:

- Certain types of optimizations can be done more easily on this form of the program.
- It can be used as a possible target language and used for *interpretive* execution, as used in Java (Bytecode), PASCAL (P-code, see Section 8.5.1), FORTH (see Section 8.6), etc.

Such an implementation structure provides a more secure language (Java) or compact translation (FORTH).

- Two versions of the compiler can be built:
  - a *check-out* or debugging compiler, which executes the user program via an interpreter (also called a *virtual machine*) of the Intermediate code, for a quick partial compilation and execution, run-time error diagnostics, execution tracing, etc.
  - an optimizing compiler, which operates further on the Intermediate code, does various forms of optimizations and generate a machine *native* code. This code will become *production* version of the program.

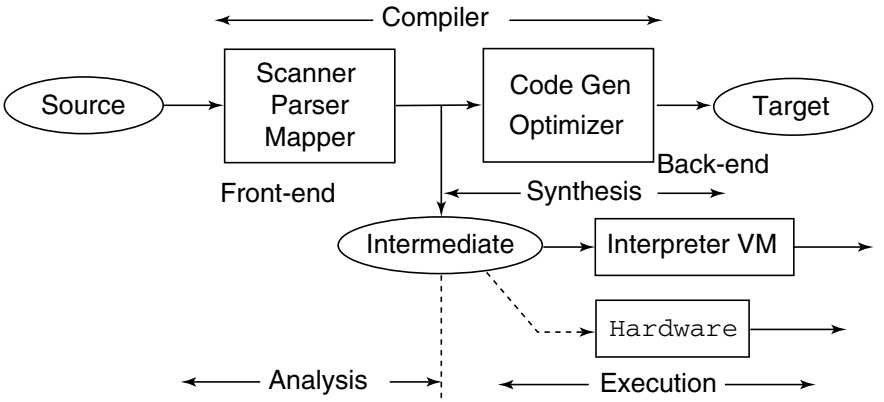


Fig. 8.1 Intermediate language form of the program being translated links the analysis and the synthesis parts

Note that the high-level language specification, which in fact is defined by the compiler, remains essentially the same in both ways of using the compiler.

- If the selected Intermediate form is machine independent, then the compiler is more *portable*.
- Ideally, for n languages and m machine architectures, total code writing effort will be $O(n + m)$ and not $O(nm)$ (see Fig. 8.2).
- **Disadvantage:** The interpretive execution will be less efficient compared to direct translation to the native machine language. This disadvantage is adequately offset by numerous advantages listed above.
- We should also note that in real-life a High Level language implementation is almost always a judicious mix of the pure compilation and interpretation schemes.
- Apart from the two major ways in which the Intermediate form is used – as a step in full translation to the target language and as language for interpretive execution – a third possibility has come up in recent years. Can we not design and build a hardware (CPU) whose machine language itself is the Intermediate form? At least partial success with Java has been achieved, which can lead to Java machines.

We keep the three possible uses of the Intermediate form in mind, while discussing it in detail.

Although C itself is used as an Intermediate language by some high-level languages (Eiffel, Sather, Esterel) which do not output object or machine code, but output C code only, to be submitted to

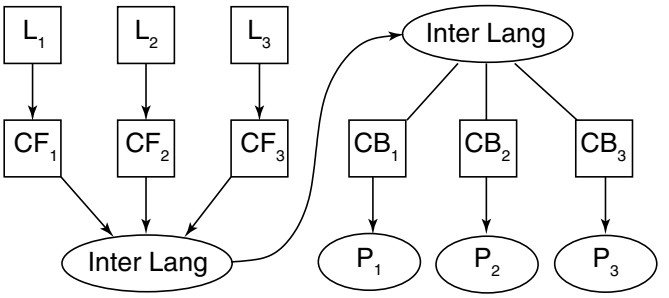


Fig. 8.2 Number of program modules required is $n + m$ for n languages and m machine architectures

a C compiler, which then outputs finished object or machine code, we restrict ourselves to more traditional intermediate codes.

We now discuss several possible Intermediate forms of the program. We also indicate how to generate the intermediate code for some of them. For that purpose we use the following example grammar, given in `yacc` input format.

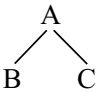
```
program   : 'P' stmt-list '.' ;
stmt-list : stmt
          | stmt ';' stmt-list ;
stmt      : expr | if | while | let ;
if        : 'I' '(' expr ')' stmt-list 'E'
          | 'I' '(' expr ')' stmt-list ':' stmt-list 'E';
while     : 'W' '(' expr ')' stmt-list 'Z' ;
let       : 'L' 'a' '=' expr ;
expr      : term | term '+' expr ;
term      : factor | factor '*' term ;
factor    : '(' expr ')' | a ;
```

Note that in order to have almost trivial Scanner, we have used single-character "keywords". Also, we do not distinguish between numerical constants and variables – all are represented by '*a*'. A close look at the grammar will tell you that we can build a recursive-descent parser for this grammar, which in fact is an extension of grammar given in Sections 1.6.2 and 2.2.2. First, we see how to get a parse tree out of the parser.

8.1 Building a Parse Tree

Suppose we want to obtain the parse tree for a program in the language generated by the above grammar. In anticipation of further processing phases, we would like to have an internal form most suitable for reduction to an Intermediate representation. However, here for illustration purpose, first we represent the parse tree in a form suitable for actually drawing the tree, which is done with the help of a TEXpackage known as `pst-tree`.

In the `pst-tree` package, a tree is written as `\pstree{root}{node1 node2 ... noden}`, where each node can be written in one of several different ways, but we use `\TR{node-label}` to write a node. For example, `\pstree{\TR{A}}{ \TR{B} \TR{C}}` gives the tree.



How do we generate the parse tree? The idea is rather simple – each Non-Terminal in the grammar should correspond to a sub-tree, with that Non-Terminal as the root-node. Each Terminal should be a leaf-node. We can put `printf()` statements in our RDP at the beginning of each function printing out the relevant portion of the total tree. For example, the following code segments show how a Non-Terminal and a Terminal output their contribution to the parse tree.

```
pT(){
     enter("\\pstree{\\TR{T}}{");
     pF();
     while(nextsymbol() == '*') {
         ppast();
         pF();
     }
     out();
}
pplus(){
     enter("\\TR{+}");
     if(symbol() != '+')
         error("+ expected");
}
```

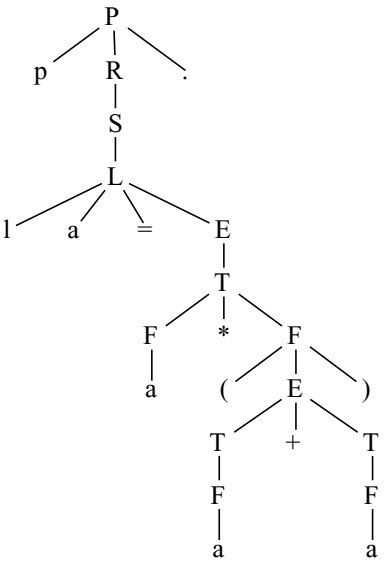


Fig. 8.3 Parse tree for input string `PLa = a * (a + a)`

The `enter()` simply prints out its argument. Function `out()` simply prints a '}'. In fact the parse tree shown in Fig. 8.5 (and others too) was obtained from an RDP in this manner. A sample RDP program `parser_tree.c` is available. For example, with an input `PLa = a * (a + a)`, a parse tree shown in Fig. 8.3 is obtained. Before we close the discussion of generation of a parse tree, in anticipation of later need, let us consider how a parse tree data structure can be generated in the computer memory.

8.1.1 Generating Parse Tree in Memory

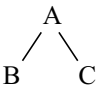
The basic idea is still the same as displaying of the picture of the tree as given above. However, while drawing the tree, conversion from the linear representation, i.e. in a serial text form, to the two-dimensional picture was done by the `pst-tree` package. To generate the tree in the memory, we have to write code for some of that work. A C program to generate the parse tree `parser_tree_tb.c` is available. We show here its essential segments.

A tree-node is defined in a header file `tb_tree_l.h` as follows:

```
typedef union{
  struct node_struct*N;
  struct tree_struct*T;
  char*S;
  long I;
  float F;
```

```
  void *P;
}util;
typedef struct node_struct{
struct node_struct *link[2]; // left = 0, right = 1
util u,v,w,x,y,z;
}Node;
```

Though this tree-node definition is optimized for a binary search tree, it is flexible enough to deal with very complex tree structure. We use pointer `u.N` as the first child pointer and `v.N` as a pointer to the next sibling. Also, we use `w.S` as a pointer to the node label string. For example, the tree



is represented in memory as shown in Fig. 8.4.

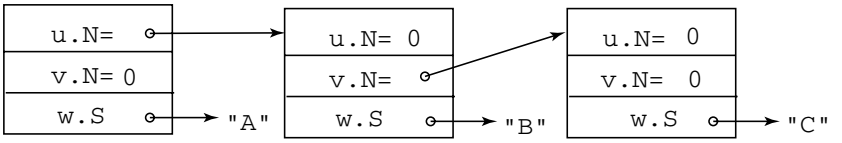


Fig. 8.4 A sample tree representation in memory

With this arrangement in mind, we write two essential functions:

```
#include "tb_tree_l.h"
Node * tree_root(char *label){
  Node *np;
  np = (Node *)malloc(sizeof(Node));
  if(np == NULL) exit(-1);
  np->u.N = NULL;          // ptr first child (if any)
  np->v.N = NULL;          // ptr next sibling (if any)
  np->w.S = (char *)malloc(strlen(label)+1);
  strcpy(np->w.S, label); // node label
  return np;
}
void add_child(Node *root, Node *child){
  Node *np;
  np = root->u.N; // children list start
  child->v.N = NULL;
  if(np == NULL){
    root->u.N = child;
    return;
  }
  while(np->v.N != NULL){
    np = np->v.N; // next in child list
  }
  np->v.N = child;
}
```

We would like to have a look at the parse tree that we created in memory, for which we write a tree traversal function:

```
void traverse_parse_tree(Node *root){
  Node *np;
  printf("[%s" ,root->w.S);
  np = root->u.N;
  while(np != NULL){
    traverse_parse_tree(np);
    np = np->v.N;
  }
  printf("]");
}
```

The following code segments illustrate modifications to the basic RDP code for generating an internal parse tree. A typical function for a Non-Terminal symbol in the grammar looks like:

```
Node *pE(){
  Node *np;
  np = tree_root("E");
  add_child(np,pT());
  while(nextsymbol() == '+') {
    add_child(np,pplus());
    add_child(np,pT());
  }
  return np;
}
```

A function for a Terminal symbol in the grammar looks like:

```
Node *psemi(){
  if(symbol() != ';') error("; expected");
  return(tree_root(";"));
}
```

With `PLa = a * (a + a)`. as input the parse tree is internally built and `main()` displays it as:

```
[P[p][R[S[L[l][a][=][E[T[F[a]][*][F[(][E[T[F[a]]][+][T[F[a]]]][)]]]]]]]
[.]]
```

Compare this output with that shown in Fig. 8.3.

With this introduction to how to build a parse tree in memory we now go on to discuss Intermediate representations.

8.2 Polish Notation

Polish notation, also known as *prefix* notation, is a form of notation for writing expressions in logic, arithmetic and algebra. The Polish logician Jan Lukasiewicz invented this notation around 1920 in order to simplify sentential logic. In this notation, the operator is written as a prefix, i.e. to the left of the

operands. If the *arity* (the expected number of operands) of the operators is fixed, the result is a syntax devoid of any parentheses that can be parsed without ambiguity. If the notation is used to represent mathematical expressions by interpreters of programming languages, they can be readily parsed into abstract syntax trees and define a one-to-one representation for the same. A typical expression in Polish notation is $\boxed{* + 3\ 4\ 5}$, which is equivalent to our normal *infix* expression (3 + 4) * 5.

Reverse Polish notation (RPN), also known as *Postfix* notation, is a mathematical notation where every operator follows all of its operands, in contrast to Polish notation. It is parenthesis-free as long as operator arities are fixed. A typical expression in RPN is $\boxed{5\ 3\ 4 + *}$, which is equivalent to our normal *infix* expression (3 + 4) * 5.

The *postfix* or *Reverse* Polish notation is used in many stack-based programming languages like FORTH, PostScript, and is the way certain calculators, notably from Hewlett-Packard, operate.

Evaluation of an RPN expression expects an operand stack. The evaluation method is shown in Algorithm 8.2.1.

Algorithm 8.2.1: Evaluation of RPN

```
while input token left do
    Read next token;
    if token is operand then
        Push token on oprnd stack;
    else
        n ← number of arguments for the operator;
        if len(stack) < n then
            print "ERROR!";
        else
            pop to n operands from stack;
            compute by the operator using n arguments;
            push result, if any, on stack;
        end
    end
end
value on top-of-stack is the result;
```

For the example RPN expression $\boxed{5\ 3\ 4 + *}$, evaluation proceeds as:

Input	Operation	Remarks	Stack
5 3 4 + *	stack	oprnd, stack	
3 4 + *	stack	oprnd, stack	5
4 + *	stack	oprnd, stack	5 3
+ *	calc	opr, pop(2), calc, push(1)	5 3 4
*	calc	opr, pop(2), calc, push(1)	5 7
	end	Result	35

The original RPN was meant to work with logical, arithmetic or algebraic expressions. We extend it to deal with programming control constructs also so that we can use it to represent Intermediate code. We call it ERPN (extended RPN). Suppose we want to represent the following `if` statement:

IF	expr	THEN	s1	ELSE	s2	FI

We can represent it in ERPN form, stored in an array, as:

0	1	2	3	4	5	6	7
expr	l1	BZ	s1	l2	BR	s2	

Here, BZ (Branch-if-Zero) is a primitive operator that causes a branch to `l1`, which is the index of the first token of `s2`, if `expr` evaluates to zero or FALSE. Similarly, BR (branch) is a primitive operation which provides an unconditional Jump to `l2`, the index of token next to the end of this IF statement. If we assume that `s1` and `s2` occupy only one array position (a rather unusual situation), then `l1 = 6` and `l2 = 7`.

Let us see how this construct can be interpreted. Scanning from left to right, first the `expr`, in RPN, is interpreted and a value, 0 or 1, is left on the stack. Next `l1`, being an operand, is put on the stack. Then BZ is scanned, which pops 2 operands, 0/1 and `l1`, and branches to `l1`, i.e. `s2`, if the `expr` is 0. That provides the ELSE branch. If `expr` were 1, no branch will take place, `s1` is scanned and interpreted. Then `l2` is stacked and operator BR will pop it to unconditionally branch to token next to the IF-THEN-ELSE statement.

Table 8.1 Examples of RPN generation – simple statements

Input string	Output RPN
`P (a + b) * c.`	`a b + c * .`
`P L s = d * (f + g).`	`d f g + * s = .`
`P (q + w) * (e + r).`	`q w + e r + * .`
`P a + b + c + d.`	`a b + c + d + .`
`P L s = d * (f + g);L g = h + j * k + l.`	`d f g + * s = h j k * + l + g = .`

A little thought will tell you that the canonical control construct WHILE-DO can be represented with the above simple arrangement. For example,

WHILE	cond	DO	s1	DONE

can be represented by:

0	1	2	3	4	5	6
cond	l1	BZ	s1	l2	BR	

Here, `l1` = 6 is the index of token next to this WHILE-DO. If `cond` is not FALSE, this branch is not taken and statement `s1` is executed. Next an unconditional branch is taken to `l2` = 0, i.e. the first token of the `cond`. Remember that the statement `s1` should be such that eventually it makes the `cond` FALSE, otherwise we will have an infinite loop.

The RPN representation is not very convenient from performing optimization, though it is used frequently to build virtual machines.

8.2.1 Generating RPN

We now see how Reverse Polish notation can be generated from a parse tree. Suppose we want the RPN for an expression `3 * (4 + 5)`. Its parse tree is shown in Fig. 8.5 and syntax tree in Fig. 8.6. The syntax tree is obtained by traversing the parse tree in *pre-order* and creating a sub-tree when an operator is encountered, with the operator as the root and the operands under it from the parse tree. The RPN can be generated by traversing the syntax tree in *post-order* and outputing nodes of each operator sub-tree. For example, the post-order traversal shown in Fig. 8.6 will give us $\boxed{3\ \boxed{4\ 5\ +}\ *} = 3\ 4\ 5\ +\ *$, which is a correct RPN. Note that we can combine the two operations and obtain the RPN directly from the parse tree.

We have a C language program `syntax_rpn_tree.c` which implements this idea for an RDP for the example grammar. Table 8.1 shows some simple examples.

The essential changes in our RDP are illustrated by the following code segments. The functions for Terminal symbols will return the symbol itself as a string, with a few exceptions as noted below.

```
char * pplus(){
    if(symbol() != '+')
       error("+ expected");
    return "+";
}
```

The functions for Non-Terminals will return the RPN built-up till now. They will build the partial RPN by post-order traversal as mentioned above. As these functions for Non-Terminals invoke one another, it is necessary to hold the partial results during each invocation separately in a stack. We could have used a separate stack for that purpose, but then we would have to write code for management of that stack. We have instead used the normal C language Return-stack itself, by reserving space for the required strings.

```
char * pT(){
     char *t2, *t3, *t4;
     char res[1024], tmp[1024];
     sprintf(res,"%s",pF());
     while(nextsymbol() == '*') {
        t2 = ppast();
        t4 = pF();
        sprintf(tmp,"%s %s %s", res,t4,t2);
        sprintf(res,"%S",tmp);
        tokens++;
     }
     t3 = res;
     return t3;
}
```

In the above code, the variable `tokens` is used for a purpose which will become clear subsequently. Each of the Non-Terminal function increments this variable by certain amount, related to the length of the partial RPN generated by it.

We also need to implement generation of RPN for at least the canonical alternate (IF-THEN-ELSE) and loop (WHILE-DO). For an IF construct, the RDP would insert I just after the condition-expression, i.e. to represent

IF cond1 THEN stmt-list1 FI

RDP would generate:

cond1-RPN I stmt-list1-RPN …

We should replace 'I' with `l1 BZ` as discussed previously, where `l1` is the index of the first token of the next statement. Also, for the full IF construct

IF cond1 THEN stmt-list1 ELSE stmt-list2 FI

the RDP would generate:

cond1-RPN I stmt-list1-RPN : stmt-list2-RPN …

Again we should replace ':' with `l2 BR`. Some examples of generated IF construct RPN are given in Table 8.2. Both these operations require that we maintain a count of number of tokens generated in the RPN and insert proper values as `l1` and `l2`, and while doing so, still keep a correct count of the RPN tokens generated. The modified RDP function which generates both short and long versions of IF construct is given below.

```
char * pI(){
  char * t1, *t2, *t3, *t4, *t5;
  char res[1024], tmp[512], tmp2[512];
  int l1, l2;
  t1 = pi(); // returns 'BZ'
  tokens +=2;
  plp();
  t2 = pE(); strcpy(tmp2,t2);
  prp();
  t3 = pR(); strcpy(tmp,t3); l1 = tokens + 2;
  if(nextsymbol() == ':'){
    t4 = pbar(); // returns 'BR'
    tokens += 2;
    t5 = pR();
    l2 = tokens;
    sprintf(res, "%s %d %s %s %d %s %s", tmp2,l1,t1,tmp,l2,t4,t5);
  } else {
    sprintf(res, "%s %d %s %s", tmp2,l1-2,t1,tmp);
```

Table 8.2 Examples of RPN generation – IF constructs

Input string	Output RPN
`P I(a + a) L s = d * (g + h)E.`	`a a + 12 BZ d g h + * s = .`
`P I(a + a) L s = d * (g + h):L x = c + vE.`	`a a + 14 BZ d g h + * s = 19 BR c v + x = .`
`PI(a * a)Ls = a + b;Lg = h * jE.`	`a a * 15 BZ a b + s = h j * g = .`
`PI(a + a) I(b * c)Ls = d * f: Lc = v + bEE.`	`a a + 22 BZ b c * 17 BZ d f * s = 22 BR v b + c = .`

```
    }
    pe();
    t5 = res;
  return t5;
}
```

Examples of generation of WHILE construct are given in Table 8.3.

Table 8.3 Examples of RPN generation – WHILE construct

Input string	Output RPN
PW(a + a)Lz = x * (c + v)Z.	a a + 14 BZ x c v + * z = 0 BR.
PW(a + a)Lz = x * (c + v); Ls = d + fZ.	a a + 19 BZ x c v + * z = d f + s = 0 BR.
PW(a + a)W(b * c)Lz = x * (c + v)ZZ.	a a + 21 BZ b c * 19 BZ x c v + * z = 5 BR 0 BR.
PLa = b; W(a)I(c)Ld = f + gEZ.	b a = a 16 BZ c 14 BZ f g + d = 3 BR.

Similar considerations hold while handling a WHILE-DO construct and we give the modified RDP function to generate a WHILE-DO construct right away.

```
char * pW(){
  char * t1, *t2, *t3, *t4, *t5;
  char res[1024], tmp[1024];
  int l1, l2;
  t1 = pw(); // returns 'BZ'
  l2 = tokens; tokens+=2;
  plp();
  t2 = pE(); strcpy(tmp,t2);
  prp();
  t3 = pR();
  t4 = pz(); // returns 'BR'
  l1 = tokens + 2;
  sprintf(res, "%s %d %s %s %d %s ",tmp, l1, t1, t3, l2, t4);
  t5 = res;
  tokens += 2;
  return t5;
}
```

You will notice that `tokens` is advanced at several places within these functions. This is to maintain the correct index values for nested constructs.

We have used absolute RPN-token index values, but many implementations of RPN generator use List-pointer relative index, i.e. the difference between the target index and the index of the Branch token.

8.3 N-tuple Notation

In this scheme of Intermediate code representation, we have a sequence of N-tuples, where $N = 3$ or 4. The first field of the N-tuple is an operator and the rest $N - 1$ fields are operands. If $N = 3$,

it is called *Triple* notation and if $N = 4$, it is called *Quadruple* notation, these two being the most popular ones.

8.3.1 Triple notation

The triple notation is also called "two-address code" (TAC). An expression $a + b$ is represented as $\boxed{+}\boxed{a}\boxed{b}$. An original expression $a * b + c * d$ would be represented as a sequence of such triples:

1.	*	a	b
2.	*	c	d
3.	+	(1)	(2)

Here, (1) denotes the result of triple 1 and (2) the result of triple 2.

A statement:

```
IF a > b THEN c = a ELSE c = b + 1 FI
```

is represented by a triple sequence:

1.	−	a	b	
2.	BMZ	(1)	5	
3.	=	c	a	
4.	BR		7	
5.	+	b	1	
6.	=	c	(5)	
7.	—	—	—	next

Code optimization is difficult with the triple notation, because optimization usually involves *movement* of the triples within the sequence. This will require changes in the sequence reference numbers within the triples. If you have done machine language programming of microprocessors, you will readily see the difficulty. In fact, the triples are like instructions of some two-address virtual machine.

One way in which the problem with the triples can be solved is to use *indirect* triples. There we use an array of pointers to the actual triples, which are not moved. If need arises during the optimization phase to move any triples, only the corresponding pointers in the array are moved.

8.3.2 Quadruple Notation

A quadruple consists of four fields: $\boxed{\text{op}}\boxed{\text{opr1}}\boxed{\text{opr2}}\boxed{\text{opr3}}$.

For example, an expression $(a + b) * (c + d) - e$ can be represented by:

1.	$+$	a	b	$T1$
2.	$+$	c	d	$T2$
3.	$*$	$T1$	$T2$	$T3$
4.	$-$	$T3$	e	$T4$

These schemes ease optimization. $T1$, etc. which represent *temporary* values, and are called temporary variables, can later be assigned to CPU registers or memory addresses, during the Code Generation phase.

8.3.3 Generation of N-tuple Code

We have already encountered the scheme for N-tuple generation while discussing the Intermediate code generation for the SIMPLE language in Section 2.2.3. We do not repeat it here.

8.4 Abstract Syntax Tree

Remember that a *parse* tree corresponds to the derivation of a sentence from the start symbol, where each internal node of the tree represents a non-terminal (NT) in the grammar and each leaf-node represents a Terminal (T). It is generated by the analysis phases – Scanner and Parser – of the compiler.

On the other hand, a *syntax* tree has as each node either some operator or some operand in the target language. A syntax tree represents a particular construct in the source language, but from the viewpoint of the target language. It is one possible output of the semantic phase, useful for optimization and Code Generation.

Sometimes we want to talk about a syntax tree having a particular structure, but in a general way, without specifying the actual operands that would appear in the target code. Then it is called an Abstract Syntax tree, where each node is either a specific (or general) operator or a token representing a data type.

An example will make the distinction clear. Consider a sentence in a simple expression grammar:

```
E -> T { + T }
T -> F { * F }
F -> ( E ) | n
```

and a sentence `3 * (4 + 5)`.

The derivation for the given sentence is (slightly condensed):

```
E -> T -> F * F -> F * (E) -> F * (T + T)
  -> n * (T + T) -> n * (F + F) -> n * (n + n)
  -> 3 * (4 + 5)
```

The parse tree is shown in Fig. 8.5.

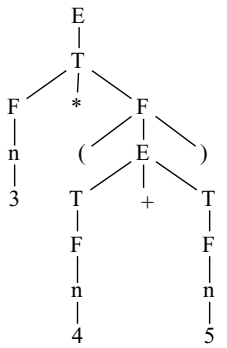


Fig. 8.5 A parse tree for the example expression

The parse tree represents the actions on the Scanner and Parser phases, but it is not very useful for the later phases such as optimization and code generation. For that we have to consider the syntax tree, shown in Fig. 8.6.

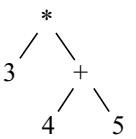


Fig. 8.6 A syntax tree for the example expression

The syntax tree conveys the *meaning* of the expression and that is what we need for optimizing and generating the target code.

Suppose we wish to discuss how to optimize or generate code for any expression having a structure $a + (b * c)$, of which the example expression $3 + (4 * 5)$ is an instance, we would use an Abstract Syntax Tree (AST), shown in Fig. 8.7.

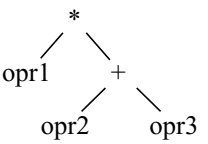


Fig. 8.7 Abstract syntax tree for the example expression

Note that the Abstract Syntax tree is a generalization of a syntax tree. Such a syntax tree (while operating on the actual code) or an Abstract Syntax tree (which discussing algorithms) can be used for optimization and Code Generation.

8.4.1 Generating Abstract Syntax Tree

We have already noted that the syntax tree can be obtained from the parse tree by traversing the parse tree in *pre-order* and a sub-tree is created when an operator is encountered, with the operator as the root and the operands under it from the parse tree. We have a C language program `syntax_tree_tb.c` which generates an AST in the memory for programs in the sample grammar. It is constructed as a modification of the basic RDP.

A few sample outputs from `syntax_tree_tb.c` are given in Table 8.4. Satisfy yourself that the AST does represent the original constructs.

Table 8.4 Examples of AST generation

Input string	Output AST
`Pa.`	`[p[a][.]]`
`Pa * (s + d).`	`[p[*[a][+[s][d]]][.]]`
`PLs = d * (f + g).`	`[p[=[s][*[d][+[f][g]]]][.]]`
`PI(a + a)Ls = d * (f + g)E.`	`[p[i[+[a][a]][=[s][*[d][+[f][g]]]]][.]]`
`PI(a)a * a:a + aE.`	`[p[i[a][:[*[a][a]][+[a][a]]]][.]]`
`PW(a)Lw = e * rZ.`	`[p[w[a][=[w][*[e][r]]]][.]]`
`PLq = w + e;a * s;d + f.`	`[p[;[;[=[q][+[w][e]]][*[a][s]]][+[d][f]]][.]]`

The tree management functions are the same as those used for generating parse tree in memory as given in Section 8.1.1; only the way that parsing function inserts sub-trees is different. As we want to generate a syntax tree, like RPN generation in Section 8.2.1, the tree construction steps are performed near the end of each of the functions for non-terminals. The functions for the Terminals supply single-node sub-trees which form leaves of the syntax tree. We present here code segments for a Terminal and a couple of Non-Terminals.

```
Node *pass(){
     if(symbol() != '=')
        error("= expected");
     return(tree_root("="));
}
```

The following function recognizes an expression and returns sub-AST for that.

```
Node *pE(){
    Node *np1, *np2;
    np1 = pT();
    while(nextsymbol() == '+') {
        np2 = pplus();
        add_child(np2,np1);
        add_child(np2,pT());
        np1 = np2;
    }
    return np1;
}
```

Function for WHILE-DO construct is:

```
Node *pW(){
   Node *np;
   np = pw();
   plp();
   add_child(np,pE());
   prp();
   add_child(np,pR());
   pz();
   return np;
}
```

8.5 Abstract or Virtual Machine Code

The Intermediate form could be "machine code" for an *abstract* or *virtual* machine. Usually, the abstract machine selected, closely models the constructs and operations of the High Level language being processed. P-code used with many PASCAL compilers and the Bytecode used by Java compilers are good examples.

The abstract machine models primitive operations and data types in a High Level language. Usually, the instruction set of the abstract machine consists of single-address instructions.

Although defining an abstract machine for a particular High Level language is relatively easy, defining a single abstract machine for a number of diverse languages, e.g. C, Java, Lisp and Prolog, is a difficult task.

8.5.1 P-code for a PASCAL Machine

The abstract machine for PASCAL is called P-machine and the code which it executes is called P-code. It is a stack machine, i.e. the instruction set is defined for data operations with respect to an operand stack. It consists of a stack, five registers and a memory (see Fig. 8.8). The five registers are explained below:

PC – Program Counter: Keeps track of the next code position.

NP – *New* pointer: Top of the Heap, location from where next dynamically, explicitly, allocated memory will be issued.

SP – Stack pointer: Top of the execution stack.

MP – Mark pointer: Base of the stack-frame of current function.

EP – Extreme stack pointer: The maximum possible size of each stack-frame is known at compile-time and this is denoted by EP. SP cannot go beyond EP.

The memory is viewed as a linear array of words and has two major parts – code and store. The stack contains a series of *stack-frames* (Activation Records, see Sections 7.1.3 and 9.6.3.) In P-code, the stack-frame contains:

- "Mark stack" part – an implicit parameter;
- a possibly empty parameter section – explicit parameters;
- local data section – explicit parameters;
- temporary elements – compiler created locals;

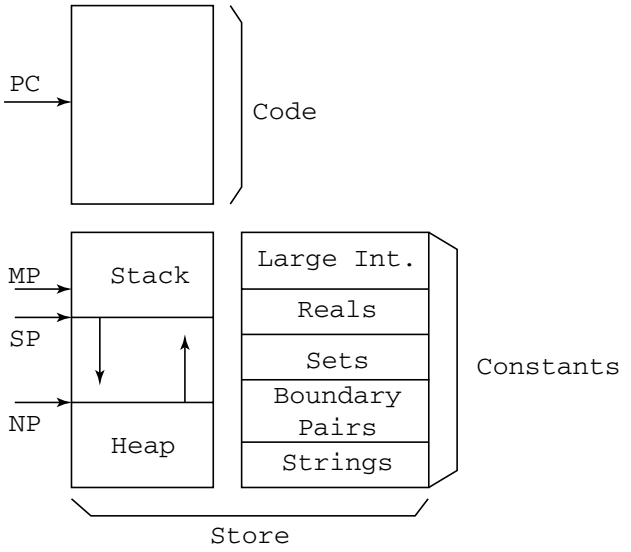


Fig. 8.8 A typical stack machine

The "Mark stack" part contains implicit parameters:

- value field for the return value from the function (not filled for a procedure);
- static link – pointer to the "Mark stack" part of the routine or the block just outside of this one;
- dynamic link – pointer to the "Mark stack" part of the calling block;
- maximum stack size value for this block;
- return address – an address within the code area.

The P-machine has several classes of instructions, like integer and real arithmetic, logical, relational, conditional and unconditional branches, subroutine call, etc. Some of them are:

ABR: Absolute value of real, pop the real on the TOS and push back its absolute value.

ADI: Adds two integers popped from the top of the stack and leaves an integer result.

DVR: Real division.

CUP: Call user procedure.

INN: Test set membership.

UJP: Unconditional jump.

8.5.2 Java Bytecode

Java Bytecode is the input language for the Java Virtual Machine (JVM). Remember that Java is a *strongly typed* language and many of the automatic upgrading and downgrading of value types are not available in it.

The JVM is expected to correctly perform operations given in a Java class file. Operations like memory layout, garbage collection and internal optimization are not part of JVM specifications and are left to the implementer.

There are several differences between Java language and the JVM language, for example Boolean in Java is represented by an integer in JVM.

Run-Time Data Areas

The following data areas are considered:

At start-up:

Heap: Shared between threads, the Garbage Collector operates on this area (need not be contiguous).

Methods area: Similar to *text* area in Unix/Linux systems.

Per thread: Created when a thread is created.

PC: A pointer to the Bytecode list. It is undefined for the *native* code, but it can hold the return address or pointer to a native code subroutine.

VM stack: Stores locals, temporaries, invocation return, etc. It is allocated on the Heap and need not be contiguous.

Run-time constants pool: Per class or interface.

Native methods stacks: Used, for example, implementation of VM in C.

Frames (AR): Activation Records (Chapter 7) created on the JVM stack. Contains an operand-stack, local variables, etc.

Instruction set of JVM consists of instructions with one-byte op-code plus zero or more bytes for operands (Big-endian).

We illustrate the nature of Java Bytecode program (i.e. contents of a class file) by a simple example of a Java program to calculate Fibonacci numbers.

```
public class fibonacci{
  static int fibonacci(int n){
    if(n > 1) return fibonacci(n - 1) + fibonacci(n - 2);
    else return 1;
  }
  public static void main(String[] args){
    for(int i = 0; i < 10; i++)
      System.out.println("fibonacci("+i+") = "+fibonacci(i));
  }
}
```

We obtained a human-readable version of the corresponding Bytecode program by a Java system utility `javap`:

```
static int fibonacci(int);
 Code:
   0: iload_0
   1: iconst_1
   2: if_icmple     19
   5: iload_0
   6: iconst_1
   7: isub
   8: invokestatic  #2; //Method fibonacci:(I)I
   11: iload_0
   12: iconst_2
```

```
   13: isub
   14: invokestatic #2; //Method fibonacci:(I)I
   17: iadd
   18: ireturn
   19: iconst_1
   20: ireturn
}
```

We have not shown the Bytecode the `main()` method. A careful scrutiny of the Bytecode will tell you that:

- `iload_0` represents the first invocation argument and it is pushed on the operand stack.
- `iconst_1` represents a constant value 1, pushed on the operand stack.
- `if_icmple 19` represents "compare the top two operands on the operand stack and jumps to instruction 19 if the first operand is less than or equal to the second".
- `isub` subtracts the TOS from the second from TOS and leaves the result on the stack.
- `invokestatic #2` calls fibonacci() method, it being the second method, the first is init() which we have not shown.
- `iadd` adds the top two operands on the stack and leaves the result there.

Thus, we see that Java Bytecode is in Reverse Polish notation and the JVM is a stack-oriented machine.

8.6 Threaded Code

This Intermediate form of the program is derived from the fact that it is possible, for many High Level languages, to express and optimize the translated form of the program as a sequence of CALLs to certain primitive functions. Just as a thread strings beads together, the "thread" of *control flow* strings these functions together to achieve the intent of the source program.

The threaded code works in *semi-interpretive* mode and is amenable to machine-independent optimization. It is the basis for implementation of languages like FORTH, PostScript, GhostScript, Java Bytecode, .NET bytecode, etc. It was also used in the FORTRAN and COBOL compilers on PDP-11 and VAX-11 machines.

There are several ways in which a threaded code can be implemented. We discuss below the most well-known among them. First, we see how the idea of a threaded code arises.

8.6.1 Subroutine Threaded Code

This is how a program translated from a typical High Level language into machine language would work (see Fig. 8.9). The execution is controlled by three operations:

NEXT: Execute next instruction:

```
fetch instruction from (PC)
PC++
execute the fetched instruction
```

CALL: Jump-to-subroutine A instruction (JSR):

```
push PC on stack
PC = address of subr A
do NEXT
```

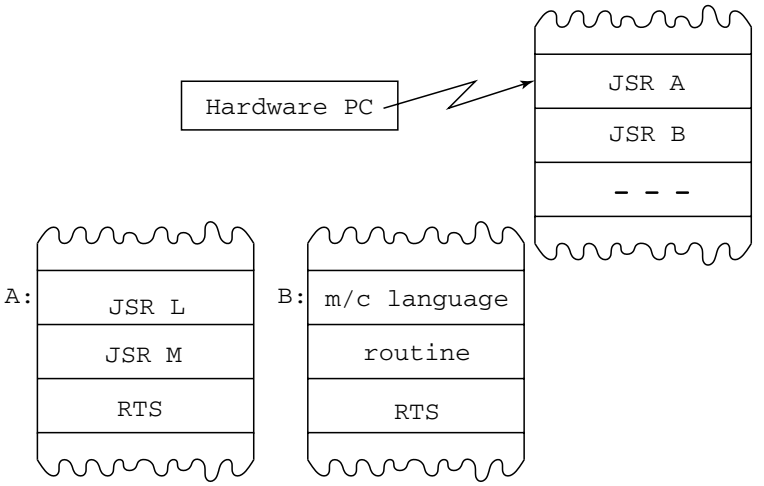


Fig. 8.9 Subroutine threaded code – beginning of threaded code idea

RETURN: Return-from-subroutine (RTS):

```
PC = pop(stack)
do NEXT
```

Note that the hardware stack, which is generally present in all modern CPUs, is used to save PC values during a subroutine CALL.

Somebody thought: "why not remove the JSR op-code? It will save bytes. Keep only the subroutine addresses" that leads us to the Direct Threaded code.

8.6.2 Direct Threaded Code

See Fig. 8.10. The three critical operations are implemented as follows:

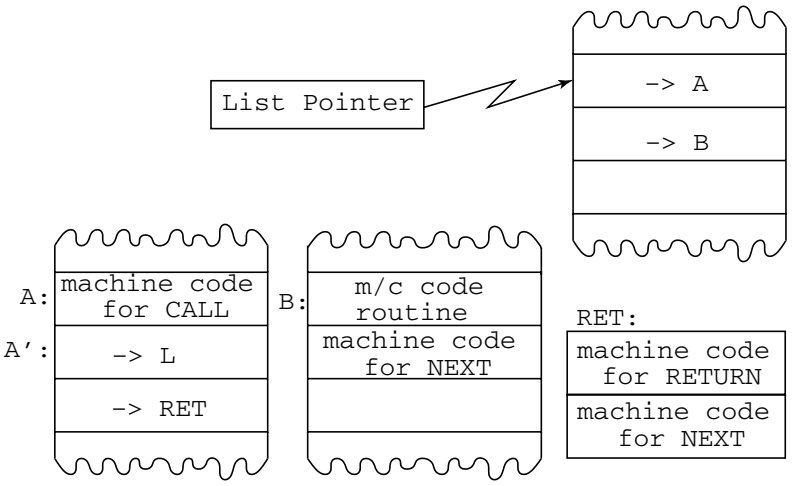


Fig. 8.10 Direct threaded code

NEXT: Execute next list item as a machine code:

```
TMP = Mem(LP)
increament LP to next item
JMP (TMP)
```

CALL: Call a subroutine A:

```
push LP to stack
LP = beginning of list-subroutine of A, A'
do NEXT
```

RET: Return from list-subroutine:

```
LP = TOS
do NEXT
```

We strongly advise you to see that you understand these three operations and convince yourself that they work for both list-subroutines and machine language subroutines.

8.6.3 Indirect Threaded Code

Indirect Threaded code is preferred on machines which have indirect addressing available at low cost (i.e., only a few clock cycles), as it gives a somewhat more compact code compared to the Direct Threaded code (see Fig. 8.11).

The three critical operations are implemented as follows:

NEXT: Execute next list item as a machine code:

```
TMP = Mem(LP)
increament LP to next item
```

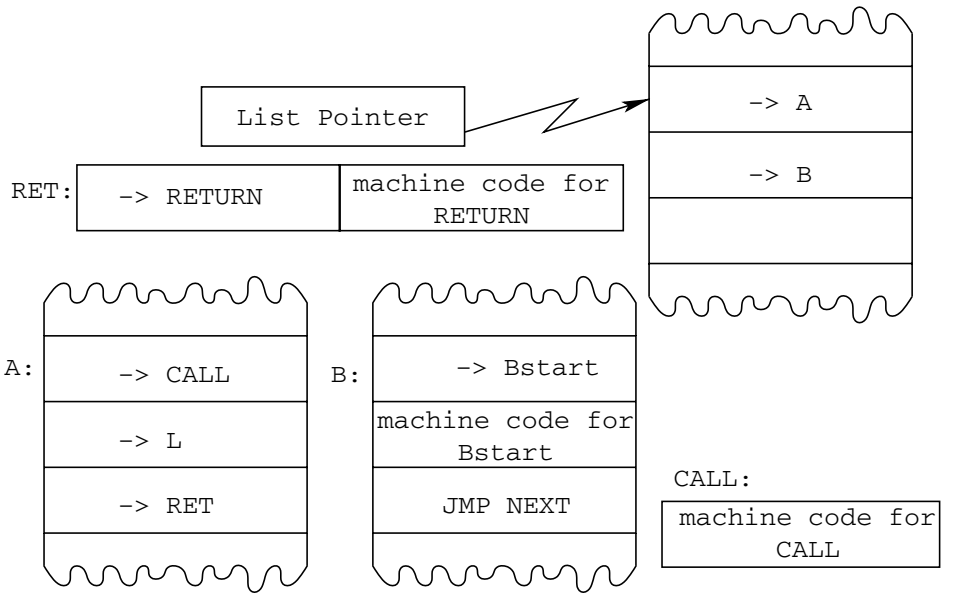


Fig. 8.11 Indirect threaded code

```
CODE-TMP = Mem(TMP)
JMP (CODE-TMP)
```

CALL: Call a subroutine *A*:

```
push LP to stack
increment TMP to next item
LP = TMP
do NEXT
```

RET: Return from list-subroutine:

```
LP = TOS
do NEXT
```

8.6.4 Token Threaded Code

Direct Threaded code cannot be simply transported from one machine to another, because it contains the code addresses that vary from machine to machine. Token threaded code uses a fixed virtual machine instruction encoding, allowing code portability at the price of a table lookup (that maps the instruction token to its code address) in each NEXT. Indirect threading and token threading are orthogonal, so they can be combined into indirect token threading (with an even slower NEXT). Usually, the token is a single byte, which allows up to 255 basic "op-codes" in the virtual machine.

8.6.5 Implementation of Threaded Code on Motorola 68000

We present here a typical implementation of Direct Threaded code on Motorola 68000 machine. The machine registers used are as follows:

PC: Normal hardware PC

SP: (A7) Return-stack pointer

PSP: (A6) Parameter stack pointer

LP: (A5) List-pointer

TMP: (A4) Temporary storage

The three critical operations are implemented as follows. Note that they are stored in the memory in the order shown, which makes for a very compact code.

```
RET: MOVE.W (A7)+, A5
NEXT: MOVE.W (A5)+, A4
     JMP (A4)
CALL: MOVE.W A5, -(A7)
     ADD.W 16, A5
     MOVE.W (A5)+, A4
     JMP (A4)
```

8.6.6 Implementation of Threaded Code on Intel x86 Machines

Here is a typical implementation of Direct Threaded code on Intel x86 series machines. The machine registers used are as follows:

PC: Normal hardware PC

SP: (EBP) Return-stack pointer

PSP: (ESP) Parameter stack pointer

LP: (ESI) List-pointer

The three critical operations are implemented as follows, in the order as stored in the memory.

```
CALL: LEA EBP, -4(EBP)
      MOVW (EBP), ESI
      ADDW EAX, 4
      MOVW ESI, EAX

NEXT: LODS
      JMP *(EAX)

RET: MOV ESP, (EBP)
     LEA EBP, 4(EBP)
     LODS
     JMP *(EAX)
```

8.7 SECD and WAM

There are two virtual machines of historical importance – SECD for the LISP and other *functional* languages and Warren abstract machine (WAM) for the PROLOG and other similar *logic* languages. Both these machines were intended to be used in three ways – for interpreted execution, further translation into a more conventional native target language of a commercial processor and for direct execution by a specialized hardware processor.

The SECD is an abstract machine with facilities and properties well suited for *functional* languages like LISP. It was invented by Pete Landin (1963) as a sample target machine for LISP. It uses a linked list oriented architecture and convenient target for S-expressions. The basic memory model of SECD consists of an array of tagged memory cells of fixed size, each having a unique address. The content of a cell can be interpreted to have several formats, by dividing it into several fields. For example, a cell tagged as "Integer" will look like:

Int	Integer Data Value

The so-called "cons" cell, used to form the linked lists, looks like:

cons	car cell addrss	cdr cell addrss

where *car* denotes the head of a list and *cdr* denotes the "tail" or remaining portion of a list. A chain of such cells defines a linked list.

SECD has four basic registers for its operation – *S* the *stack* register points to a list in memory which behaves like stacks in conventional computers. *E* the *environment* register points to the current value list of function arguments. *C* the *control* register, works like the Program Counter of a conventional computer. It points to the memory cell which contains the current instruction to be executed. Instead of incrementing, it progresses through the cdr chaining. *D*, the *dump* register remembers the current states of *S*, *E* and *C* registers when a function calls itself recursively. It behaves very similar to call–return sequence in a conventional computer.

The origin of WAM is in the PhD thesis of David H. D. Warren (1977). The basic architecture is Von Neumann-like, though there is a strong dependence on a stack. A variety of procedure call instructions directly implement the sequencing philosophy assumed by the PROLOG inference engine as it cycles through the *goal* lists and the program *clauses*. Other instructions provide a wide variety of *Unification* tests.

8.8 Grammar and IR Generation for `miniC`

In Chapter 12, we plan to present a complete development of a compiler for a stripped down C-like language, `miniC`. Here we present, as examples of Intermediate Representation for most important program constructs, along with the action terms added in the `yacc` grammar to generate them. The following are considered:

- expressions
- assignments
- statements: IF-THEN, IF-THEN-ELSE, WHILE-DO
- IF-THEN and IF-THEN-ELSE initiation
- WHILE-DO initiation
- variable declarations
- function definitions
- function calls

We use three additional stacks to keep track of various code-related parameters:

tstack: Stack for the temporaries `Tn` number, `tcount` used in the 4-tuple IR. Corresponding stack functions are `tpush()` and `tpop()`. The 4-tuple IR assumes "infinite" number of temporary locations available.

mstack: Stack for the 4-tuple entry numbers, counted by `mcount`. Generally, this count goes on incrementing, but we need to stack its value while generating the control constructs like IF-THEN-ELSE or WHILE-DO. The corresponding stack manipulation functions are `mpush()` and `mpop()`.

lstack: Stack for *label* numbers created for generating IR for control constructs. The label count is maintained by `lcount`. The corresponding stack manipulation functions are `lpush()` and `lpop()`. The labels are used by the generators for both RPN IR and 4-tuple IR.

You will notice that the generation of 4-tuple IR is slightly more complex than the generation of RPN IR. This is so because 4-tuple is nearer to the assembly or machine language of most of the modern processors, and almost one-to-one correspondence can be established between 4-tuples and a short sequence of assembly or machine instructions. On the other hand, mapping the RPN to assembly or machine instructions involves use of a stack to store operands, especially for mapping expressions.

One important difference in the action code from the one given in Chapter 5 is that, instead of putting all the outputs in one output file, we now output the RPN IR in file `fout1` and the 4-tuple matrix in file `fout2`. The file names are created out of the source file name.

8.8.1 Expressions

We had already presented IR generation for a few of the possible expressions in Chapter 5. Here we see some more details. In case of RPN, we expect that the IR generated for the two operand sub-expressions leave their values on the top of the stack; we simply generate a code for the operator.

On the other hand, for 4-tuple IR, we generate an entry by popping the top two `tcount` from the `stack` and assigning next `tcount` to the result. Then we should push the result `tcount` on the `tstack`.

```
expr: INT { $$ = $1; fprintf(fout1,"%s\n", name($1));
            fprintf(fout2,"%d: LD %s T%d --\n",++mcount,name($1),
                                        ++tcount);tpush(tcount);}
    | NUMBER { $$ = $1; fprintf(fout1,"%s\n", name($1));
               fprintf(fout2,"%d: LD %s T%d --\n",++mcount,name($1),
                                        ++tcount);tpush(tcount);}
    | VAR    { $$ = $1; fprintf(fout1,"%s\n", name($1));
               fprintf(fout2,"%d: LD %s T%d --\n",++mcount,name($1),
                                        ++tcount);tpush(tcount);}
    | IVAR   { $$ = $1; fprintf(fout1,"%s\n", name($1));
               fprintf(fout2,"%d: LD %s T%d --\n",++mcount,name($1),
                                        ++tcount);tpush(tcount);}
    | expr '+' expr { fprintf(fout1,"+\n"); $$ = $3;
      fprintf(fout2,"%d: ADD T%d T%d T%d\n",++mcount,tpop(), tpop(),
                                        ++tcount); tpush(tcount);}
    | expr '-' expr { fprintf(fout1,"-\n"); $$ = $3;
      fprintf(fout2,"%d: SUB T%d T%d T%d\n",++mcount,tpop(), tpop(),
                                        ++tcount); tpush(tcount);}
    | expr '*' expr { fprintf(fout1,"*\n"); $$ = $3;
      fprintf(fout2,"%d: MUL T%d T%d T%d\n",++mcount,tpop(), tpop(),
                                        ++tcount); tpush(tcount);}
    | expr '/' expr { fprintf(fout1,"/\n"); $$ = $3;
      fprintf(fout2,"%d: DIV T%d T%d T%d\n",++mcount,tpop(), tpop(),
                                        ++tcount); tpush(tcount);}
```

The basic assignment operation is rather straightforward, but `miniC` contains `op=` type of assignments also. The semantics of these extended assignments raises an issue. The normal subtract operation, written in infix notation as `a -b`, in RPN appears as `a b -` and the semantics is "subtract the TOS from one below it". The semantics of extended assignment `a -= e` is "subtract the value of `e` from `a`. The corresponding RPN IR that we would generate is `e a - a =`, but it reverses the order of subtraction. We have two options – either exchange the top two values on the stack, or use a special form of – operator, shown as `R-`(reversed minus), which has semantics "subtract from TOS the value below it". Similarly, divide operator also being non-commutative, we use `R/` for "reverse divide". The 4-tuple IR has no such issues.

8.8.2 Assignments

```
asgn: VAR '=' expr  {fprintf(fout1,"%s\n=\n", name($1));
                     fprintf(fout2,"%d: = T%d %s --\n", ++mcount,tcount,
                                          name($1));tpush(tcount);}
    | IVAR '=' expr {fprintf(fout1,"%s\n=\n", name($1));
                     fprintf(fout2,"%d: = T%d %s --\n", ++mcount,tcount,
                                          name($1)); tpush(tcount);}
```

```
| IVAR ADDEQ expr {fprintf(fout1,"%s\n+\n%s\n=\n", name($1),name($1));
      fprintf(fout2,"%d: ADDE T%d %s %s\n", ++mcount, tcount, name($1),
                                        name($1));tpush(tcount);}
| IVAR SUBEQ expr {fprintf(fout1,"%s\nR-\n%s\n=\n", name($1),name($1));
      fprintf(fout2,"%d: SUBE T%d %s %s\n", ++mcount, tcount, name($1),
                                        name($1));tpush(tcount);}
| IVAR MULEQ expr {fprintf(fout1,"%s\n*\n%s\n=\n", name($1),name($1));
      fprintf(fout2,"%d: MULT T%d %s %s\n", ++mcount, tcount, name($1),
                                        name($1));tpush(tcount);}
| IVAR DIVEQ expr {fprintf(fout1,"%s\nR/\n%s\n=\n", name($1),name($1));
      fprintf(fout2,"%d: DIVE T%d %s %s\n", ++mcount, tcount, name($1),
                                        name($1));tpush(tcount);}
```

We illustrate by a small example:

```
int : a,b
a = 5
b = a - 3
a - = 2
a = a - 2
end
```

The generated RPN IR is:

```
int a DCL int b DCL 5 a = a 3 - b = 2 a R - a = a 2 - a =
```

and the 4-tuple IR is:

```
DCL int a --
DCL int b --
1:  LD 5 T1 --
2:  = T1 a --
3:  LD a T2 --
4:  LD 3 T3 --
5:  SUB T2 T3 T4
6:  = T4 b --
7:  LD 2 T5 --
8:  - a T5 a
9:  LD a T6 --
10: LD 2 T7 --
11: SUB T6 T7 T8
12: = T8 a --
```

8.8.3 Statements

A statement can be a variable declaration, a function definition, an expression, a function return, a WHILE-DO or an IF-THEN construct. Because certain initialization actions are needed for many of them, the IR generation for a complex control construct is divided into two parts. Explanation of code generation is explained in each type of statement.

```
stmt:    decl
       | defn
       | expr
       | RETURN  {fprintf(fout1,"RET\n");
                  fprintf(fout2,"%d: RET T%d -- --\n", ++mcount, tcount);
                                                      tpush(tcount);}
       | RETURN expr {fprintf(fout1,"RET\n");
                  fprintf(fout2,"%d: RET T%d -- --\n", ++mcount, tcount);
                                                      tpush(tcount);}
       | while '(' cond ')' stmt end {int l; l = lpop();      /* label2 */
                  fprintf(fout1, "Label%d\nBR\n", lpop());
                  fprintf(fout1, "Label%d\n:\n", l);
                  fprintf(fout2, "%d: BR %d -- --\n", ++mcount,($1)->x.I);
                  fprintf(fout2, "%d: LABI %d -- --\n", ++mcount,
                                                      ($3)->y.I);}
       | if '(' cond ')' stmt end {                   /* else-less if */
                  fprintf(fout1, "Label%d\n:\n", lpop()); lpop();
                                                      /* label1 */
                  fprintf(fout2, "%d: LABI %d -- --\n", ++mcount,
                                                      ($3)->y.I);
                  }
       | if '(' cond ')' stmt end ELSE {int l,l1 ;
                  l = lpop();                         /* label2 */
                  fprintf(fout1, "Label%d\nBR\n", (l1=lpop())); lpush(l1);
                                                      /* label1 */
                  fprintf(fout1, "Label%d\n:\n", l);     /* label2 */
                  fprintf(fout2, "%d: BR %d -- --\n", ++mcount, 0);
                                                  ($3)->x.I = mcount;
                  fprintf(fout2, "%d: LABI %d -- --\n", ++mcount,
                                                      ($3)->y.I);
                  }
                  stmt end {
                    fprintf(fout1, "Label%d\n:\n", lpop());
                                                      /* label2 */
                    fprintf(fout2, "%d: LABI %d -- --\n", ++mcount,
                                                      ($3)->x.I);
                  }
       | '{' stmtlist '}'         { }
       ;
```

Examples of the code generation are given with each type of statement.

8.8.4 IF-THEN and IF-THEN-ELSE

The problem with the control constructs is that we have to generate a reference to a remote point in the generated code, depending upon the condition that prevails at run-time. In case of RPN, this can be a symbolic label, like `Label1`, which is comparatively easy to manage. Basically, the operation similar to what we did in the case of Recursive-descent Parser (RDP) will work for `yacc`-based parser also. However, for 4-tuple IR generation, we need to know the 4-tuple entry number for a remote branch, which is not known due to intervening statements.

Note that if we were generating the assembly code for a particular processor directly, then also we could have simply generated symbolic labels for remote jumps, but we want to investigate the machine-independent optimization, which requires that IR form of the program be available.

Starting with IF-THEN construct, where we need to have action like:

{compute logical expr } {BranchFalse Label1: } {execute stmts } {Label1: . . .

With IF-THEN-ELSE, we need to have action like:

{compute logical expr } {BranchFalse Label1:} {execute stmts-1} {Branch Label2:}

{Label1: stmts-2 } {Label2: . . .

In the first case, within the construct, we need to generate a reference to one remote location `Label1`, in the second case we need to have two references generated `Label1` and `Label2`, but we do not know where they will fall in the final generated IR code. In case of RPN, we have to output these labels in the output stream at appropriate point in the generated code. In the case of 4-tuple, the code generator needs actual remote 4-tuple entry number.

We decide to use a symbol called "dope" as a semantic value assigned to the keyword token `if`. We use it as a means of communication between various action terms. For each IF construct used in the source program, a fresh "dope" symbol is generated. We also generate two label numbers and push them on the `lstack`, by the action terms shown below. IF-THEN will use only one of them and discard another one, IF-THEN-ELSE will use both of them.

```
if: IF { Symbol *i; i = tree_root("dope"); $$ = i; ++lcount; lpush(lcount);
                                                   ++lcount; lpush(lcount);}
```

Now comes the interesting part. On detection of a `cond` in the IF or WHILE construct, we pop the first label, and in RPN we generate a "Branch-if-Zero Label1" code in the output stream and push back that label on `lstack`. Later, referring to Section 8.8.3, when the end of the true statements is detected, we send out the first label and discard the second.

```
cond:  expr {int l; fprintf(fout1, "Label%d\nBZ\n", l=lpop()); lpush(l);
            ($$)->y.I = ++mcount;
            fprintf(fout2, "%d: BZ %d -- --\n", mcount, 0); }
       ;
```

In case of IF-THEN-ELSE, again referring to Section 8.8.3, we generate and output a "Branch-to Label2" code at the end of the true statements and generate and output Label1 to mark the position in the RPN where control should go if the `cond` tests false. When the end of the false statements is detected, we pop the second label and output it.

If we were generating the 4-tuple IR, the situation is slightly more complex. As noted above, we need to have the remote entry numbers as the branch targets. This really requires two passes over the source code – in the first pass we note the remote points to which a conditional or unconditional branch may take place and the second pass we generate the IR code. This is somewhat similar to

a two-pass assembler. Although it is possible by an additional stack, which stores generated code for the "statements", to achieve this in a single pass, we decide to work as follows. We generate the 4-tuple IR for the complete source program, generating branch entries with unknown branch target, and taking care to note the entries within it which correspond to the remote branch targets. At the end of the IR generation, we go through this 4-tuple matrix and insert correct entry numbers in the branch 4-tuples.

Thus when the `cond` symbol is detected, the next 4-tuple entry number is saved in a field within the $$ symbol, as a note to remember where we have to insert correct 4-tuple number later, and a 4-tuple is generated and output for branch to 0. Let us say the entry number is `i`. In the IF-THEN, action on detection of the end of statements, we generate a 4-tuple with a special operation `LABI`, with the entry number saved in $$ symbol as the first argument. Let us say that its entry number is `j`, which looks like `j : LABI i -- --`. The post-processing label-insert function will detect this LABI code, insert `j` in entry number `i` and change LABI to LAB.

Similar actions are performed for the IF-THEN-ELSE construct; the only difference is that for the second half of the IF construct, another field in the `cond` symbol is used to store the entry number of the ELSE part.

For an example source code using partial, IF-THEN construct:

```
int : a,b,c,d
if(a = 3) d = a * (b + c) end
end
```

we obtained the following IR (RPN):

```
3 a = Label2 BZ a b c + * d = Label2 :
```

and the following 4-tuple matrix:

```
1:  LD 3 T1 --
2:  = T1 a --
3:  BZ 0 -- --      3: BZ 10 -- --
4:  LD a T2 --
5:  LD b T3 --
6:  LD c T4 --
7:  ADD T3 T4 T5
8:  MUL T2 T5 T6
9:  = T6 d --
10: LABI 3 -- --   10: LAB 3 -- --
```

For an example source using full IF-THEN-ELSE:

```
int : a,b,c,d
if(a = 3) c = 7 else d = a * (b + c) end
end
```

we got the following IR (RPN):

```
3 a = Label2 BZ 7 c = Label1 BR Label2 : a b c + * d = Label1 :
```

and the following 4-tuple IR:

```
1:  LD 3 T1 --
2:  = T1 a --
3:  BZ 0 -- --        3: BZ 7 -- --
4:  LD 7 T2 --
5:  = T2 c --
6:  BR 0 -- --        6: BR 14 -- --
7:  LABI 3 -- --      7: LAB 3 -- --
8:  LD a T3 --
9:  LD b T4 --
10: LD c T5 --
11: ADD T4 T5 T6
12: MUL T3 T6 T7
13: = T7 d --
14: LABI 6 -- --     14: LAB 6 -- --
```

8.8.5 WHILE-DO

In our `miniC` grammar, the `cond` symbol is used in both IF and WHILE constructs. WHILE will need to branch to the beginning of the IR code for the condition computation that is why the WHILE-DO construct is initialized differently.

```
while:   WHILE {Symbol *w; fprintf(fout1, "Label%d\n:\n", ++lcount);
                lpush(lcount); ++lcount; lpush(lcount);
                w = tree_root("dope"); w->x.I = ++mcount; $$ = w;
                fprintf(fout2, "%d: LAB %d -- --\n", mcount, mcount);
                mpush(mcount);}
       ;
```

For the generation of RPN IR, we generate two labels, push them on the `lstack` and output the first label. Referring to Section 8.8.3, when the "statements" end is detected, we generate a branch to the first label and issue the second label in the output. The BZ instruction in the action term of `cond` token will branch out of the WHILE loop if condition is not satisfied.

For the generation of 4-tuple IR, we create and assign a symbol `w`, which records the 4-tuple entry number at the beginning of the `cond` in its `x` field, and issue a label to the output.

When the end of the "statements" is detected, a branch to this 4-tuple is issued and then a `j : LABI i -- --` 4-tuple is issued, where `j` corresponds to entry number of this 4-tuple and `i` refers to the entry number of the `i: BZ 0 -- --` 4-tuple, which will be updated by post-processing, as we explained for IF construct.

For an example source code shown below:

```
int : a,b,c,d
while(a = 3) d = a * (b + c) end
end
```

we got the following IR (RPN):

```
Label1 : 3 a = Label2 BZ a b c + * d = Label1 BR Label2 :
```

and the following 4-tuple IR:

```
1:  LAB 1 -- --
2:  LD 3 T1 --
3:  = T1 a --
4:  BZ 0 -- --          4: BZ 12 -- --
5:  LD a T2 --
6:  LD b T3 --
7:  LD c T4 --
8:  ADD T3 T4 T5
9:  MUL T2 T5 T6
10: = T6 d --
11: BR 1 -- --
12: LABI 4 -- --      12: LAB 4 -- --
```

Note that the IR generator code output the BZ command at matrix line 4 with a zero branch "address". Later at line 12, LABI command sets a label and also indicates that the line number of the matrix line where this label is set should be inserted in line 4 and then LABI command is changed to LAB. These line address operations are carried out after the whole of the IR is obtained. Though it is possible to write an IR generator which does this "on-the-fly" in one pass, it involves a separate additional stack. We chose the simpler method shown here. We also tried nesting an IF construct within a WHILE loop:

```
int : a,b,c,d
while(a = 3) if(b = 5) d = a * (b + c) else d = 9 end end
end
```

we got the following IR (RPN):

```
Label1 : 3 a = Label2 BZ 5 b = Label4 BZ a b c + * d = Label3 BR
Label4 : 9 d = Label3 : Label1 BR Label2 :
```

and the following 4-tuple IR:

```
1:  LAB 1 -- --
2:  LD 3 T1 --
3:   = T1 a --
4:  BZ 0 -- --          4: BZ 20 -- --
5:  LD 5 T2 --
6:  = T2 b --
7:  BZ 0 -- --          7: BZ 15 -- --
8:  LD a T3 --
9:  LD b T4 --
10: LD c T5 --
11: ADD T4 T5 T6
12: MUL T3 T6 T7
13: = T7 d --
14: BR 0 -- --         14: BR 18 -- --
15: LABI 7 -- --       15: LAB 7 -- --
```

```
16: LD 9 T8 --
17: = T8 d --
18: LABI 14 -- --            18: LAB 14 -- --
19: BR 1 -- --
20: LABI 4 -- --             20: LAB 4 -- --
```

8.8.6 Variable Declarations

Declaration statements for variables are mainly concerned with transmission of the *inherited* attributes to the Symbol Table entry for each of the variables being declared. Also, the same information has to be passed on in the RPN and 4-tuple IR being output. See Chapter 6 and Sections 5.4, 5.4.4 and 5.4.7 for details of the *synthesized* and *inherited* attributes.

```
decl:    dtype ':'  dlist {}
       ;
dtype:   IVAR
       | VAR
       | SVAR
       ;
dlist:   VAR     {type($1) = type($<sym>0); subtype($1) = type($<sym>0);
                  fprintf(fout1,"%s\n%s\nDCL\n",tokname(subtype($1)),
                                                          name($1));
                  fprintf(fout2,"DCL %s %s --\n", tokname(subtype($1)),
                                                         name($1));}
       | dlist ',' VAR { type($3) = type($<sym>0); subtype($3) = type($<sym>0);
                         fprintf(fout1,"%s\n%s\nDCL\n",tokname(subtype($3)),
                                                          name($3));
                         fprintf(fout2,"DCL %s %s --\n", tokname(subtype($3)),
                                                         name($3));}
       ;
```

Examples of IR generated for variable declarations are already included in other examples.

8.8.7 Function Definitions

Function definitions in `miniC` have syntax:

```
func <func-name> () statement
```

In the version of `miniC` presented here, in the interest of simplicity, the number and types of the formal arguments are not taken into account by the compiler. A function may be called with any reasonable number of arguments and because the Caller-Cleans-Stack type calling conventions are used, the run-time stack remains valid. The symbol `procname` carries with it initially the type `VAR` and name from the Symbol Table. First, we have to set the type as `FUNCTION`.

In the case of RPN IR generation, we simply output `name DFN` in the output stream. Then the actions associated with the `stmt` will output the corresponding RPN. Then we output `DFE` to indicate "definition end".

In case of 4-tuple IR generation, the definition starts with `i: DFN name -- --` and ends with `j : RET Tm -- --` where `Tm` is the temporary last assigned any value within the function.

```
defn:   FUNC procname { type($2) = FUNCTION; fprintf(fout1,"%s\nDFN\n",
                                                          name($2));
                fprintf(fout2,"%d: DFN %s -- --\n", ++mcount, name($2));
                                                     ($2)->u.I = mcount;}
           '(' ')' stmt {fprintf(fout1,"DFE\n");
                fprintf(fout2,"%d: RET T%d -- --\n",++mcount, tcount);}
      ;
procname: VAR
        | FUNCTION
        ;
```

With the following example source code:

```
int: a, b, c
b = c + 9
func myfunc() a = a + 7 end
c = 11
```

we got the following IR (RPN):

```
int a DCL int b DCL int c DCL c 9 + b = myfunc DFN a 7 + a = DFE
```

and the following 4-tuple IR:

```
DCL int a --
DCL int b --
DCL int c --
1: LD c T1 --
2: LD 9 T2 --
3: ADD T1 T2 T3
4: = T3 b --
5: DFN myfunc -- --
6: LD a T4 --
7: LD 7 T5 --
8: ADD T4 T5 T6
9: = T6 a --
10: RET T6 -- --
```

8.8.8 Function Calls

A function call (invocation) in `miniC` looks like: `funcname(arg1, arg2, ..., argn` and is expected to return a value if used on the RHS of an assignment or within an expression.

For RPN IR generation, each `expr` in the argument list will leave its value on the stack, so a function call consists of only issuing the name of the function. The function is expected to pick-up correct number of call arguments from the stack, compute and leave the result on the stack.

For 4-tuple IR generation, first we issue a function call by a 4-tuple as: `i: CALL funcname Tm --` where `Tm` is the temporary holding the function return value. After that, we issue arg 4-tuples as: `j : ARG Tp -- --` where `Tp` is the temporary holding the result from the respective argument expression. The following example will clarify these concepts:

```
expr | FUNCTION begin '(' arglist ')' { fprintf(fout1,"%s\n", name($1));
                fprintf(fout2,"%d: CALL %s T%d --\n", ++mcount, name($1),
                                                        ++tcount);}
arglist: /* nothing */    {$$ = 0; }
       | expr             { fprintf(fout2,"%d: ARG T%d -- --\n",++mcount,
                                                        tpop()); $$ = 1; }
       | arglist ',' expr { fprintf(fout2,"%d: ARG T%d -- --\n",++mcount,
                                                        tpop()); $$ = $1+1;}
       ;
expr   | ARG { fprintf(fout2,"%d: CARG %d T%d --\n",++mcount,4 * ($1 + 1),
                                                ++tcount); tpush(tcount);}
```

Within the function, the call arguments can be accessed as $1, $2, etc. which are detected by the Scanner and returned as ARG. We generate an entry for each of them as CARG. With the following example source input:

```
int: a, b, c
b = c + 9
func myfunc() a = a + 7
myfunc(a + 2, b - 3)
end
```

we got the following IR (RPN):

```
int a DCL int b DCL int c DCL c 9 + b = myfunc DFN a 7 + a = DFE a 2 + b 3 - myfunc
```

and the following IR (4-tuple):

```
DCL int a --
DCL int b --
DCL int c --
1:  LD c T1 --
2:  LD 9 T2 --
3:  ADD T1 T2 T3
4:  = T3 b --
5:  DFN myfunc -- --
6:  LD a T4 --
7:  LD 7 T5 --
8:  ADD T4 T5 T6
9:  = T6 a --
10: RET T6 -- --
11: LD a T7 --
12: LD 2 T8 --
13: ADD T7 T8 T9
14: ARG T9 -- --
15: LD b T10 --
16: LD 3 T11 --
17: SUB T10 T11 T12
18: ARG T12 -- --
19: CALL myfunc T13 --
```

8.8.9 Utility Functions Used

```
int tcount = 0;
int lcount = 0;
int mcount = 0;
char *tokens[] = {"float","int","string","UNDEF", 0};

char *tokname(int tok){
  return strdup(tokens[tok-VAR]);
}

Symbol * tree_root(char *label){
  Symbol *np;
  np = (Node *)malloc(sizeof(Node));
  if(np == NULL) exit(-1);
  np->link[0] = NULL; // ptr first child (if any)
  np->link[1] = NULL; // ptr next sibling (if any)
  np->w.S = (char *)malloc(strlen(label)+1);
  strcpy(np->w.S, label); // node label
  return np;
}
```

8.9 Real-life: Intermediate Codes of GNU gcc

GCC – GNU compiler collection (previously expanded as GNU C compiler), gcc/ccl – the modern GNU C compiler (see Fig. 8.12).

The Abstract Syntax Tree (AST), generic Intermediate representation in GCC:

- Output by the parser.
- Explicitly represents all information in the original source code.
- Front-end code may have its specific AST, but generic is aimed at being the common IR of all HLLs that GCC supports.

GIMPLE is another abstract level of representation in GCC. It is influenced by the SIMPLE Intermediate representation of McCat compiler. In fact, the name GIMPLE is given due to this similarity with SIMPLE. It is a subset of the AST-generic. There are two "flavours" of GIMPLE:

Structured: Cleanups and simplification.

Unstructured: Lowering of control flow, i.e. a program is represented as sequenced statements plus unrestricted jumps. Note that lowered control flow is nearer to register machines (hardware CPU) and leads to easier generation of static single assignment (SSA) form, where every variable is assigned in only one place.

GIMPLE grammar is available (see GCC reference):
`http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`

Internal data structure used to store the GIMPLE nodes is a tree and that is the reason why the intermediate output from a GCC compiler is called trees.

Tree is the central data structure of the gcc front-end. A tree-node is capable of holding integers, reals, complex, strings, etc. Actually a tree is a pointer type, but the object to which it points may vary. If we are just taking the example of an integer node of a tree, it is a structure containing an integer value, some flags and some space for RTL (these flags are common to all the nodes). There are a large

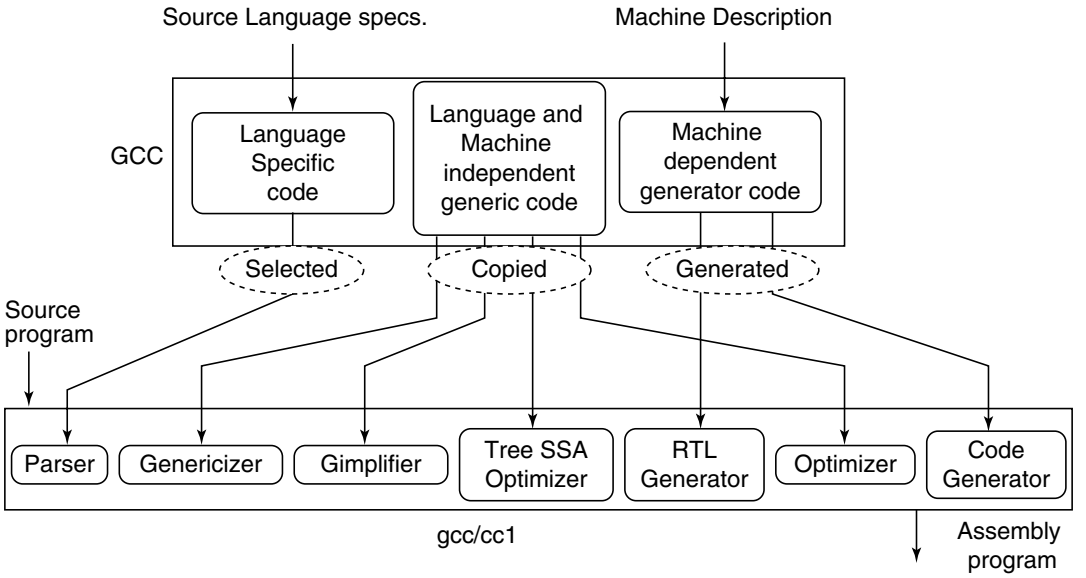


Fig. 8.12 GNU compiler collection framework. Generic, Gimple and RTL are the Intermediate codes in GCC

number of functions and macros provided by the GCC, which help us to manipulate the tree. In our program, we need not directly handle the trees. Instead we use the function calls.

Some of the GIMPLE node types are:

Binary operator – BIT AND EXPR, MAX EXPR
Comparison – EQ EXPR, LTGT EXPR
Constants – INTEGER CST, STRING CST
Declaration – FUNCTION DECL, LABEL DECL
Expression – MODIFY EXPR, ADDR EXPR, BIND EXPR
Reference – COMPONENT REF, ARRAY RANGE REF
Statement – EXIT EXPR, GOTO EXPR
Type – BOOLEAN TYPE, INTEGER TYPE
Unary – ABS EXPR, NEGATE EXPR, NOP EXPR

RTL stands for register transfer language. It is the intermediate code handled by GCC. Each and every tree structure that we build has to be changed to RTL so that the back-end can work with it perfectly. As with the trees, GCC provides a number of functions to produce the code from the trees. So while developing a compiler we are trying to build the tree and convert it to RTL for each and every line of the program. Most of the RTL routines take the trees as arguments and emit the RTL statements.

8.9.1 Example GCC Intermediate Code

In order to make concrete the above ideas, we give below the intermediate representation from a `gcc` compilation of a sample C program. These outputs were obtained by the command:

```
gcc -fdump-tree-all testintermediate.c
```

First, let us have a look at the original C source:

```
int main(){
  int i = 2, j = 0;
  if(i > j) j = i + 3;
  while(j < 10){
    i = j * 2;
    j++;
  }
  if(i == j) j = i + 3; else i = j/3;
 return 0;
}
```

The program is not written to do any useful work, but just to investigate various structures generated in the Intermediate code. Note that we have not tried any optimization while compiling.

GCC processes the parsed version of the source program, i.e. the `Parse tree`, by a series of passes which generate various forms of trees representing the original program and going nearer and nearer to what is required by the code generator. Before the code generator takes over, these trees are further processed into a form called RTL (Register Transfer Language), which is summarized in Chapter 9.

The first tree is called the original, which is a top-level representation:

```
;; Function main (main)
;; enabled by -tree-original
{
  int i = 2;
  int j = 0;
  <STATEMENT_LIST>
}
```

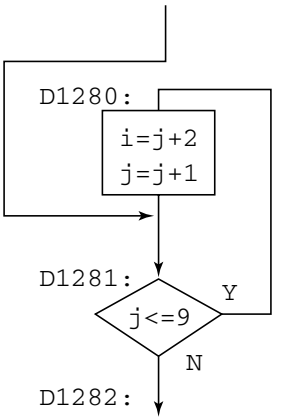
Note that the tree is output in a human-readable form, which looks like C code. If there were any functions defined in the source program, they would have been included here.

Now we have a look at the GIMPLE form:

```
main(){
  int D.1283;
  int i;
  int j;
  i = 2;
  j = 0;
  if(i > j){
     j = i + 3;
  } else {
  }
  goto <D1281>;
  <D1280>:;
  i = j * 2;
  j = j + 1;
  <D1281>:;
```

```
  if(j <= 9){
     goto <D1280>;
  } else {
     goto <D1282>;
  }
  <D1282>:;
  if(i == j){
     j = i + 3;
  } else {
     i = j / 3;
  }
  D.1283 = 0;
  return D.1283;
}
```

There are several points to be noted. `<D.1283>` represents the constant '0' which is returned by the `main()` function. The first `if` statement generates a null `else` clause. Particularly note how the `while` loop is represented by a combination of `if` and `goto`: Compare this construct with Polish notation in Section 8.2 and the N-tuple notation in Section 8.3. This is an example of "lowering the control construct".



We then have a look at one more tree, called "cfg" tree:

```
;; Function main (main)
main(){
  int j;
  int i;
  int D.1283;
<bb 0>:
  i = 2;
  j = 0;
  if(i > j) goto <L0>; else goto <L1>;
<L0>:;
  j = i + 3;
```

```
<L1>:;
  goto <bb 4> (<L3>);
<L2>:;
  i = j * 2;
  j = j + 1;
<L3>:;
  if(j <= 9) goto <L2>; else goto <L4>;
<L4>:;
  if(i == j) goto <L5>; else goto <L6>;
<L5>:;
  j = i + 3;
  goto <bb 8> (<L7>);
<L6>:;
  i = j / 3;
<L7>:;
  D.1283 = 0;
  return D.1283;
}
```

This form is the last tree for this simple program, without compiler optimization, before the RTL form is generated. Notice that now code blocks are identified and marked.

? EXERCISES

Several of the following exercises are to be done on a computer system. A Linux system is recommended. You should have `gcc` and Java Development Kit (JDK) installed.

1. The function `traverse_parse_tree()` in Section 8.1.1 traverses the parse tree in *pre-order*. Modify it to traverse in *in-order* and *post-order*.
2. In Section 8.2.1 on generating RPN, we saw that certain local values were to be stored in the functions for the Non-Terminals and we used the return-stack for the purpose. This decision was really a hack. It may work in demo programs, but in real-life code, which will be required to handle source programs of arbitrary size, the return-stack will get heavily loaded and may put a limit on the size of the programs which can be translated. The tree-node structure used has a number of `util` pointers available. Investigate how to use these pointers to associate required information with each node in the tree.
3. Find out how to obtain the intermediate byte code for a Perl program.
4. Use `javap` to obtain human-readable Bytecodes for the following Java program and explain the output as far as you can.

   ```
   public class Hello{
     public static void main (String[]args){
       System.out.println ("Hello, World!");
     }
   }
   ```

   You will have to compile the program first with the Java compiler `javac`.
5. Use `javap` to obtain human-readable Bytecodes for the following Java program and explain the output.

```
public class HelloThread extends Thread{
  public int i = 0;
  public void run(){
    i++;
    System.out.println ("Hello from a thread! "+i);
  }
  public static void main (String args[]){
    (new HelloThread()).start();
    (new HelloThread()).start();
  }
}
```

6. Use `gcc` to obtain its Intermediate representation outputs for the following sample C program:

```
int func1(int i){
  return i * (i + i);
}
int main(){
  int j = 5, k;
  k = func1(j);
}
```

Comment on your results.

FURTHER READING

1. Richard Stallman, "GNU Compiler Collection Internals", book available online at `http://gcc.gnu.org/onlinedocs/`
2. Richard M. Stallman and the GCC Developer Community, "Using the GNU Compiler Collection".
3. Sreejith K. Menon, "GCC Front-end HOWTO" by. Available at: `http://www.linuxdoc.org/HOWTO/GCC-Frontend-HOWTO.html`

GLOSSARY

AST See "Abstract Syntax Tree".

Abstract Syntax Tree A data structure representing something which has been parsed, often used as a compiler or interpreter internal representation of a program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax.

Activation record also called "data frame" or "stack-frame": A data structure containing the variables belonging to an instance of invocation of one particular scope, e.g. a procedure or function body, as well as links to other activation records.

Assembly language A symbolic representation of the machine language of a particular processor. Assembly language is converted to machine instructions by an assembler. Usually, each line of assembly code produces one machine instruction.

Bytecode The language of Java Virtual Machine.

Call instruction for invocation of a function or procedure.

Code Program.
Declaration Tells the compiler that an identifier is representing a particular object in the program.
Direct Threaded Code A technique for implementing a virtual machine or interpreters, where each op-code in the virtual machine's instruction set is the address of code to perform the required operation. If we express the inner interpreter operation as C code: `ip` is the threaded code instruction pointer; `cfa` (code-field address) corresponds to the execution token and points to the code field of the next word to be executed. `ca` (code address) fetched from there points to some executable code, e.g. a primitive or the threaded code. Then the NEXT for direct threading is simply:

```
ca = *ip++;
goto *ca;
```

Expression A syntactic construct which defines arithmetic, Boolean or relational sequence of operations.
Extended regular expression Regular expressions written with enhanced facilities beyond the canonical *catenation*, *alternation* and *closure*. Perl provides a very reach facilities for ERE.
Extended BNF Extended Backus-Naur Form.
Function A subroutine which will return a value to the calling program.
GCC GNU compiler collection, consisting of front-ends for several languages like C, C++, Fortran, Ada, Objective-C, Java and back-ends for a large number of processing platforms.
Garbage collection The process by which dynamically allocated storage is reclaimed during the execution of a program. The term usually refers to automatic periodic storage reclamation by the garbage collector – a part of the run-time system, as opposed to explicit code to free specific blocks of memory.
Generator Several meanings in computer science: Code generator – a program which creates a source program output; a system that creates source code automatically; a program which produces a stream of data; a specialized routine that acts like an iterator.
Heap memory An area of memory used for dynamic memory allocation, where blocks of memory are allocated and freed in an arbitrary order and the pattern of allocation and size of blocks are not known until *run-time*.
Heap data structure programming has its elements partially ordered such that finding either the minimum or the maximum, but not both, of the elements is computationally inexpensive, independent of the number of elements, while both adding a new item and finding each subsequent smallest/largest element can be done in $O(\log n)$ time, where n is the number of elements.
Indirect Threaded Code has one level of indirection added to Direct Threaded Code. Each op-code is now a pointer to memory where the address of the code is stored. Expressed as C code, an indirect threaded NEXT looks like:

```
cfa = *ip++;
ca = *cfa;
goto *ca;
```

Intermediate Between the front-end and the back-end of a compiler.
Interpreter Executes a program in HLL, by analyzing each statement every time it is to be executed.
Label An identifier assigned to a statement as a reference.
LISP A list-processing language.
List A data structure where non-contiguous data elements – nodes – are linked by pointers. Each node, except the last, has a next node.

Machine Any hardware, software or abstract entity which has finite number of states, accepts input in some language, does state-to-state transitions and may generate some output.
Non-Terminal symbol is an element of the meta-language *M* used to write the grammar of a language *L*, and not part of *L*. In other words, non-terminals do not appear in the final sentences generated.
Notation A technical system of symbols used to represent special things.
Operations Actions of the processor which transform data.
Operator Symbols denoting operations.
Optimization Making the target language program as comparable to one written by an experienced assembly language programmer as possible.
Procedure A subroutine which may not return any result to the calling program.
Push Put a value on top of the stack.
RPN See Reverse Polish notation
Recursive-descent parser A *top-down* parser built from a set of mutually recursive procedures, where each such procedure usually implements one of the productions of the grammar. Thus, the structure of the resulting program closely mirrors that of the grammar it recognizes.
Reference Denoting an object in a program by some kind of pointer and not directly by its identifier.
Register Elementary storage elements within a processor.
Register transfer language An intermediate code for a machine with an infinite number of processor registers, used for machine-independent optimization.
Return Taking the program control back to the calling program, possibly along with a *returned value*.
Reverse Polish notation Post-fix notation, where the operator is written immediately after its operands.
Run-time At the time of execution of a program.
Scanner Lexical analyzer identifies language atom in the source program and converts them to tokens.
Semantic Related to meaning, i.e. what are the specified actions, of the program.
Source language Language in which a human programmer prepares his/her specification of the work to be done by the computer.
Stack Last-in-first-out store. It has two basic operations push – put a value on TOS and pop – remove the value at the TOS.
Subroutine A code segment which will be written once but used a number of times by invocation.
Syntax Representation of a program in terms of grammatical structure rather than its meaning.
Target language The language to which translation is taking place.
Terminal symbol Symbol occurring in a programming language.
Thread Path of program control flow.
Token A basic, grammatically indivisible unit of a language such as a keyword, operator or identifier.
Top-of-stack Also written as TOS, the topmost and immediately accessible element on the stack.
Tree A data structure, a directed acyclic graph, i.e. a graph wherein there is only one route between any pair of nodes, and there is a notion of "towards top of the tree", i.e. the root-node, and its opposite direction, towards the leaves.
Type Value type taken by an identifier, e.g. *number*, *integer*, `string`, etc.
Unconditional jump/branch Breaking the normal "next" sequence of program operation and jumping to a specified address, without testing any conditions.

9

Code Generation and Machine-dependent Optimization

What you will learn in this chapter

- What are the issues in Code Generation?
- How does the Target language influence code generation?
- How does the Instruction Set of the processor influence code generation?
- How are the registers of the processors utilized?
- How are the registers of the processors allocated?
- How is the instruction ordering selected?
- How are various constructs in the High Level programming language implemented in the target language?
- How are Functions and Procedures implemented?
- What is Optimization?
- What are Machine-dependent and Machine-independent optimizations?
- What are the methods of Machine-dependent optimization?

Key Words

code generation, intermediate representation, target language, instruction set, registers, data structures, control constructs, procedures and functions, activation record, code optimization, machine-dependent optimization, register allocation

The last phase of compilation is Code Generation. This phase takes the Intermediate Representation of the original source program as input and produces a semantically equivalent program in the target language as output. Depending upon the nature of the target language, there may be subsequent steps necessary to make the program ultimately executable on a particular platform. In a typical case of Linux/Unix-like platform, the target language is generally a (simplified) assembly code, and further steps of assembly through the system assembler linking with language library and loading are required to get an executable program. Further, while generating the target code, it will be highly desirable to

generate a "good" code, one which satisfies the basic requirement of any good algorithm – *it should be correct and efficient*.

The basic requirements of the Code Generation phase are:

1. The generated target language program must correctly and faithfully express the algorithm specified by the source program (and not what the programmer *intended*, unfortunately!).
2. The target language program should be efficient, which means, it should use the platform resources as efficiently as possible. Target instructions most economical in CPU time utilization should be used and program layout should use minimum possible memory, consistent with that requirement.
3. The code generator itself should be efficient, and it should generate the target code as fast as possible.

In order to satisfy these requirements, several things need to be done:

1. The nature of the target code and Run-Time environment should be known well enough.
2. The factors needed to be considered in code generation – the input Intermediate Representation (IR), selection of instruction sequence in the target code, allocation of processor components, especially the CPU registers and actual order of computation in the final target code – should be examined from the viewpoint of code *Optimization*.
3. In fact, it is found that the code optimization, though conceptually a separate phase, actually has to be done in several passes over the IR and ad-hoc generated code. Thus a good, optimizing compiler will be working some what like what is shown in Fig. 9.1.

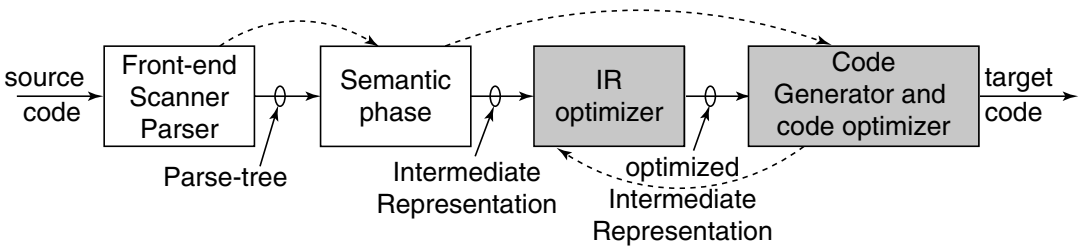


Fig. 9.1 Code generator in the compiler chain. The dashed arrows show the order in which we discuss the phases, from pedagogical necessity

We first discuss, after detailing our concerns in the task of code generation, a naïve code generator, without any worry of inefficiency of the generated code. Of course, correctness of the generated code must be ensured even in presence of special cases and idiosyncrasy of the target language. Then we point out some typical inefficiencies and see how the code generator can deal with them and generate better target code, by what is called *machine-dependent optimization*. We shall also discuss some concepts like *Basic Blocks* which are used in a real-life compiler, viz., GNU `gcc`.

9.1 Our Concerns in Code Generation

While designing a code generator, we are concerned about the following areas which form a framework in which the code generator will have to perform.

9.1.1 Input Intermediate Representation (IR)

The main inputs to the code generator are the IR from the semantic phase and contents of the Symbol Table. By the time the code generator accesses it, the Symbol Table has stored the *type* information for each variable and expression, worked out by the type-checking portion of the semantic phase.

There are several possible IRs, which we have already discussed in Chapter 5. The most useful are Reverse Polish Notation (RPN), 4-tuple (three-address) code (we call it a code-matrix) and Abstract Syntax Tree (AST).

9.1.2 Nature of Target Language

The instruction set of the target language is the most significant factor which has influence over the effort required to generate a "good" target program. There are two kinds of processor architecture from the instruction set viewpoint – Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC).

CISC is characterized by comparatively few CPU registers, some of them may be designated for special purpose, generally one- or two-address instructions, a large variety of memory addressing modes, variable length instructions (giving high code packing density) and partial or non-orthogonal instructions.

RISC is characterized by a large number of CPU registers, generally three-address instructions, only a few addressing modes, fixed length instructions and almost orthogonal instructions.

By *orthogonality* of instructions we mean that most of the instructions such as Load, Store, Move, Add, Sub, Comp, etc. will accept any of the registers and any data size or type with the same semantics. A highly orthogonal instruction set will allow easier target code optimization (see Fig. 9.2).

The instruction set may be designed for general register to register operation or it can be stack-based operations. Some years back stack-based architecture was quite popular and recently it is again gaining favour due to possibility of better and easier portability of the machine level language programs.

Though generating an absolute machine language program as a target seems attractive, as it can directly be loaded in the memory, without any further processing, this is rarely done, except possibly in the case of small embedded systems. Generally, it is preferred to generate *object* modules or files and later link them together in the form of the final executable. This is a very flexible arrangement and it also leads to code reuse.

If we generate an assembly language output – possibly in a simplified assembly language – and use the system assembler to do the assembly after the compilation, we get several advantages. First, the final address assignment to the variables can be done by the assembler and the code generator phase is relieved of that responsibility; second, the macro facility of the assembler can be used to define the target code patterns for specific IR patterns. This second aspect even goes further, for CPU differing slightly in their architecture can share all of the compilers including the code generator and just by changing the pattern macros library we can get the target code for different CPU. However, it is true that some possibilities of the machine-dependent optimization will be lost in that scheme.

9.1.3 Selection of Alternatives from Instruction Set

If the IR is low level and is nearer to the target language and target machine architecture, then it is rather simple to generate efficient target code. For example, if IR is RPN and the target machine is also a stack-based machine, then it is simple to generate the target code. On the other hand, with IR

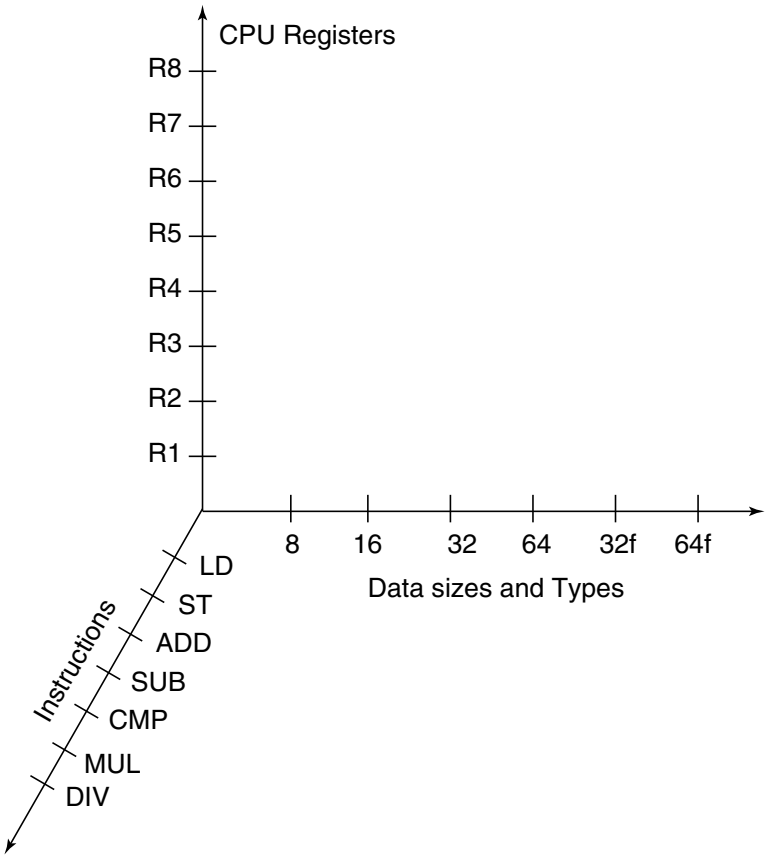


Fig. 9.2 Three "dimensions" of orthogonality of instruction set of a processor. If this 3-D cube is filled densely with points, the processor has more orthogonal instruction set

being RPN and the target machine a CISC with general-purpose registers, it requires more effort to define the mapping from IR to the target code.

We have already commented on the orthogonality of the instruction set of the target processor. There is thinking among some schools of design that the computation with floating point numbers should be executed by load and store from the same general-purpose registers as the integers and the instruction to do floating point ADD, SUB, etc. should have exactly same semantics as those for the integers. There is another group of schools which differ on this aspect due to the following reasons.

Floating point numbers are not reals as understood in mathematics. Many properties (e.g. arithmetic laws) that reals have and that one expects all kinds of numbers do not hold for floating point numbers. For example, within a computer program, it is not proper to compare two floating point variables for equality of values. Also, the way in which floating point numbers are used in a typical application is quite different from the way integers are used. Floating point numbers are used in numerical computation limited to ADD, SUB, MUL, DIV and various numerical functions like trigonometric, logarithms, exponentiation, special functions, vectors and matrix operations, etc. Code Generation

for all such computations will benefit from stack-based computation regime, as the problem of CPU register allocation (see Section 9.8.1) is resolved.

We can conclude like this: if the size of the general registers is made such that it can easily contain the floating point representation on the platform and instructions are provided for movement of floating point numbers from general-purpose registers to the floating point computation stack and vice-versa, then it can be considered an acceptable enough orthogonal design of the instruction set.

We should remember that the instruction set for each CPU type is designed with certain code *idioms*, i.e. sequence of instructions to represent some computation patterns occurring very frequently in mind. For example, in x86 series machines, a function just on entry is expected to execute the following two instructions to set up the stack-frame:

```
pushl  %ebp
movl   %esp, %ebp
```

With awareness of such idioms and ignoring the execution time taken of each instruction, it is quite easy to select the instructions from the instruction set of the target processor, for each of the possible 4-tuple instruction in the IR. Let us see where does this lead us to. Consider a small program segment given below:

```
a = 3
b = 5
c = 7
d = a * (b + c)
```

A naïve semantic phase generated the following 4-tuple code (matrix):

```
1:   LD    3   T1   --
2:   =    T1    a   --
3:   LD    5   T2   --
4:   =    T2    b   --
5:   LD    7   T3   --
6:   =    T3    c   --
7:   LD    a   T4   --    // redundant, a is already in T1
8:   LD    b   T5   --    // redundant, b is already in T2
9:   LD    c   T6   --    // redundant, c is already in T3
10:  ADD  T5   T6   T7
11:  MUL  T4   T7   T8
12:  =    T8    d   --    // redundant, put d in (11)
```

The Tn are temporaries assigned by the semantic phase, without worrying about where they will be in the actual executing program. Each of the original statement has resulted in a small, stylized, sequence of 4-tuples. At least 4 rows in the 4-tuple IR output can be removed as they are redundant.

Let us remove all the redundancies from the IR:

```
1:   =    3    a   --
2:   =    5    b   --
3:   =    7    c   --
4:   ADD  b    c   T1
5:   MUL  a   T1    d
```

What we have done is to *optimize* the IR, the process is termed *machine independent optimization* and is the subject matter of Chapter 10. Now suppose we "code generate" this IR into our target code, the language for which we have selected, for this example, the assembly language of x86. We decide to use an idiom

```
movl   p, %eax
addl   q, %eax
movl   %eax, r
```

for a three-argument operation $r = p + q$, which can be directly expressed as a 4-tuple `ADD p q r`. We can use any other binary operator in place of ADD. Note that the *assignment* is built into the 4-tuple notation, but stand alone assignment $r = p$ can use a simpler idiom

```
movl   p, %eax
movl   %eax, r
```

Now we set about the code generation for the given IR, looking at each of the five rows of the optimized IR and inserting appropriate operands in the idiom for each of the 4-tuples.

```
movl   $3, %eax
movl   %eax,  a
movl   $5, %eax
movl   %eax,  b
movl   $7, %eax
movl   %eax,  c
movl   b,  %eax
addl   c,  %eax
movl   %eax, T1
movl   a,  %eax
imull  T1, %eax
movl   %eax,  d
```

There are several ways in which the code generated above can be made more efficient. We can select a better *instruction sequence* for assigning an integer constant, instead of the two-instruction sequence we used blindly. That better idiom, indicated by closer study of the instruction set, is `movl $3, a`. This will remove 3 instructions from the above target code.

We could select for the 4-tuple `ADD p q r`, the idiom

```
movl   p, r
addl   q, r
```

again by closer scrutiny of the instruction set. Then the generated code could be

```
movl   $3,    a
movl   $5,    b
movl   $7,    c
movl   b,     T1
addl   c,     T1
movl   T1,    %eax
imull  a,     %eax      // due to nature of imull instruction
```

We could have used CPU register `eax`, in place of T1, and reduced one more instruction. That would be a small example of *machine-dependent* optimization.

Operations like `i = i + 1` is very frequent in normal programs, and most of the CPUs provide `incr` instruction for efficiently coding them. If the IR is made to contain a unary operator INC instead of `i = i + 1`, the code generation is straightforward.

9.1.4 Allocation of CPU Registers

Even the very simple examples shown above should have told you that decision about which value is to be held in which register can become quite tricky. Apart from limitations on special-purpose registers, proper allocation of registers can make the code more efficient. You should remember that registers are at the top in memory hierarchy in a computer; they are the fastest storage units accessible to the processing units and they are always in the short supply. Also, for most of the processors, the instructions operating on registers are shorter and faster (because they are shorter and deal with fast registers).

The allocation of registers has to take into account not only the current 4-tuple, but what further operations take place on the variables involved in the next few 4-tuples. Let us take an example. The following 4-tuple

```
ADD a b t
```

would be translated to

```
movl a, t
addl b, t
```

If the next 4-tuple is

```
MUL t c t
```

The generated code cannot be

```
imull c, t
```

as the `imull` instruction expects Reg/Mem/Imm as the first (src) operand and Reg as the second (dst) operand. This means that the generated instruction should be something like

```
imull c, %reg
```

However, then we better change our decision about the first 4-tuple and generate code for the two 4-tuple as:

```
movl   a, %eax
addl   b, %eax
imull  c, %eax
movl   %eax, t
```

Note that the multiplication instruction always leaves the result in the coupled registers edx:eax.

Further, suppose the next 4-tuple is

```
DIV t d t
```

that is $t = t/d$. We shall have to use the `idiv` instruction, which expects the dividend in edx:eax and divisor as an operand, which can be a register or memory. The generated code now becomes

```
movl   a, %eax
addl   b, %eax
imull  c, %eax
movl   %eax, %edx
idiv   d
movl   %edx, t
```

9.1.5 Order of Evaluation Sequence

In a long numerical expression, we may find that some particular execution order results in a more efficient target code, at least in terms of register usage. Most of the modern CPUs are pipelined, i.e. execute several instructions in parallel. This opens up a possibility of considering the evaluation order in depth, on the one hand, and the correct computation must be achieved, on the other hand; reordering of instructions will result in faster code. We postpone the discussion of this last aspect of code generation till Section 9.8.2.

9.2 The Target Language

We are going to use x86 assembly language for our project in Chapter 12. Here, we use simple subset of that language.

9.2.1 x86 Assembly Language in GAS Syntax

We use the AT&T assembly syntax used in GNU assembler `as`, also known as, GAS. The assembly code output from the `gcc` compiler uses this syntax and is compatible with the GCC in-line assembly syntax. However, this is not the only syntax that is used to represent x86 operations. For example, Net-Assembler NASM uses a different syntax to represent assembly mnemonics, operands and addressing modes, as do some High Level assemblers. The AT&T syntax is the standard on Unix-like systems but some assemblers use the Intel syntax used in MASM. GAS itself can accept both the syntaxes. We shall use only the AT&T syntax.

GAS instructions generally have the form

```
<mnemonic op-code> <source>, <destination>
```

which is the reverse of other assemblers like MASM and NASM. For example, the following `mov` instruction

```
movw $0xFEED, %ax
```

will move the hex value FEED (16-bit) into the register `AX`.

In GAS, the op-codes are generally suffixed with the letters '*b*', '*s*', '*w*', '*l*', '*q*' or '*t*' to specify the size of the operand being manipulated.

b = byte (8-bit)
s = short (16-bit integer) or single (32-bit floating point)
w = word (16-bit)
l = long (32-bit integer or 64-bit floating point)
q = quad (64-bit)
t = 10-bytes (80-bit floating point)

If the suffix is not specified, and there are no memory operands in the instruction, GAS infers the operand size from the size of the destination register operand (the final operand).

When referencing a register, the register name needs to be prefixed with a '%'. Numerical constants need to be prefixed with a '$'.

Address operand syntax Up to 4 parameters of an address operand are presented in the memory address operand syntax:

```
<displacement>(<base register>, <offset register>, <scalar multiplier>)
```

This is equivalent to:

[BaseRegister + displacement + OffsetRegister * ScalarMultiplier]

in Intel syntax. Either or both of the numeric, and either of the register parameters may be omitted. Some examples are:

`movl -4(%ebp,%edx,4),%eax`	Full: load *(ebp − 4 + (edx * 4)) into eax
`movl -4(%ebp), %eax`	Typical: load a stack-variable into eax
`movl (%ecx), %edx`	No offset: copy value at memory address in ecx into edx
`leal 8(,%eax,4), %eax`	Arithmetic: multiply eax by 4 and add 8
`leal (%eax,%eax,2), %eax`	Arithmetic: multiply eax by 2 and add eax (i.e. multiply by 3)

To summarize:

To generate a good code for the x86 series of machines we need:

- To select appropriate instructions.
- To make good use of registers and avoid unnecessary memory accesses.
- Must not use registers that are already in use.
- To keep the number of jumps as less as possible, as too many of them will reduce the CPU cache' performance.

The design decisions that we have to take are:

- How will the values of types in the source language be mapped to values of types in the target language?
- How will the constructs of the source language be mapped to constructs in the target language?
- How do we represent source language values in the target language?
- At present we shall consider only a few primitive types:

  integers → 32-bit words

  booleans → 32-bit words, 0 = false, 1 = true
- A *statement* specifies an action: So compilation of a statement should produce code to execute that action.
- An *expression* specifies a value: So compilation of an expression should produce code that can be executed to calculate that value.

To make this work we need a general strategy and a mapping of that strategy onto our target machine.

A possible general strategy is like this. Assume that we have only two registers, `eax` and `ebx`. We have to decide where to store the intermediate results. In general, it is not possible to store them in registers, because enough of them are not available. We can store them in memory, but memory exchanges take time and we have to allocate space for them. We can store them on the run-time stack, i.e. push a value on to the stack when we need to save it so that we can release a register, and pop the value off later when we need to use that saved value.

Our general strategy for compiling code will be: calculate a value for the expression and leave the result in `eax` and push `eax` on the stack to save its value when necessary. For example, to compile the expression $e1 + e2$:

```
generate code to evaluate e1, leaving result in eax
pushl %eax
generate code to evaluate e2, leaving result in eax
popl %ebx
addl %ebx, %eax
```

In short, the result of any sub-expression computation is kept in `eax`, and its value is saved on the stack when it must survive the computation of another sub-expression.

A more formal description of this code generation strategy is as follows.

```
E[e1 + e2]  =  E[e1]
               pushl %eax
               E[e2]
               popl  %ebx
               addl  %ebx, %eax
```

We can extend the generation scheme E to other forms of expression:
Constant load:

```
E[n]        =  movl $n, %eax
```

Generation of conditional logic operators:

```
E[e1 && e2] =      E[e1]
                   cmp1 $1, %eax
                   jnz      lab1       a new label
                   E[e2]
            lab1:           ...
```

Generation of variable references:

```
E[var]      =  movl var, %eax
```

For example, consider computation of an expression (2 + 4) * (3 + 5) using the stack to save the intermediate values:

```
movl   $2,%eax
pushl  %eax
movl   $4,%eax
popl   %ebx
```

```
add     %ebx,%eax
pushl   %eax
movl    $3,%eax
pushl   %eax
movl    $5,%eax
popl    %ebx
add     %ebx,%eax
popl    %ebx
imull   %ebx,%eax
```

If we use register allocation instead of the stack, because we have more than two registers:

```
movl    $2, %eax
movl    $4, %ebx
addl    %ebx, %eax
movl    $3, %ebx
movl    $5, %ecx
addl    %ecx, %ebx
imull   %ebx, %eax
```

We can modify our code generation schemes to make good use of machine registers. Further improvements are also possible by making a better use of the instruction set:

```
movl    $2, %eax
addl    $4, %eax
movl    $3, %ebx
addl    $5, %ebx
imull   %ebx, %eax
```

This just requires a specialized version of our revised generation scheme:

```
E1[e1 + n] = E1[e1]
             addl $n, %eax
```

9.3 Data Structures

We now discuss how data structures, especially vectors, arrays and records or constructs like `struct` in C are handled by the code generator. Two aspects of these code constructs are to be considered – the layout of the structure in the memory and run-time access code generation.

9.3.1 Vectors and Arrays

These structures are characterized by having elements of exactly the same size and type, arranged in a linear sequence of consecutive memory addresses. That makes the code generation relatively simple. The elements in the vector or array could be of primitive data types like `char`, `int`, `float`, `double`, a pointer or it can be a user-defined data type, which may have any arbitrary size.

9.3.2 Vectors

A vector *V* of elements `e` of type *T* is stored in the memory as consecutive groups of sizeof(*T*) bytes. On some platforms, in order to increase the speed of memory access, sizeof(*T*) may be adjusted to the

next higher multiple of 8- or 16-bytes, but we shall ignore that complication here. Figure 9.3 shows how a vector of integers is laid out and its access. From Fig. 9.3, we see that to access any element *V*[*i*] of a vector *V*, the address is `base+i*s` where *s* is the size of the elements in bytes. The total memory area to be assigned is *n* * *s* bytes, where *n* is the size of the vector. This can be easily generated as:

```
.section .bss
  .lcomm V, 32
```

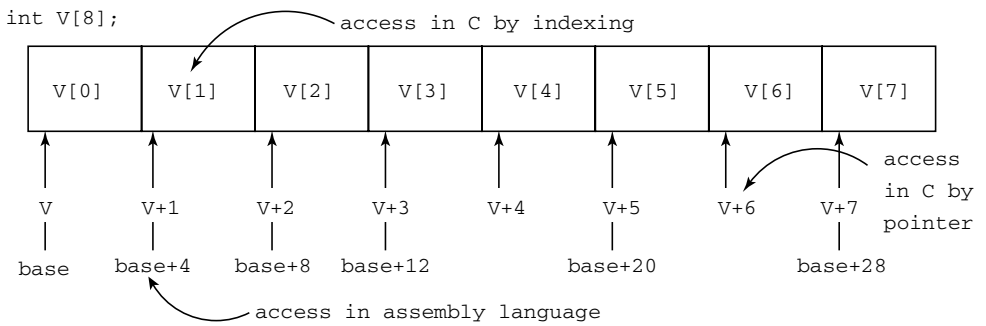


Fig. 9.3 A vector of integers and three versions of access to its elements

The access code is also simple, assuming that `%esi` contains the index (0 based) of the element we want:

```
movl V(,%esi,4), %eax
```

The value will be put in `%eax`.

In general to access *A*[*i*]:

```
base(A) + i * sizeof(A[1])
```

Depending on how *A* is declared, base(*A*) may be

- an offset from BP,
- an offset from some global label, or
- an arbitrary address.

The first two and sizeof(*A*[1]) are compile-time constants. For primitive data types, the sizeof(*A*[1]) is generally a power of 2 – 1 for `char/byte`, 2 for `short`, 4 for `int`, pointers and `float` and 8 for `double` and `long`. In x86 processors, the multiplication by the size is built into the instruction. On processors where this is not the case, it can be done by left-shift operations. For arrays consisting of elements of user-defined data types (usually structures), actual multiplication has to be used to generate the address. Even there, in case of x86 processors, `leal` instruction can be used to do the multiplication, as shown in Section 9.2.1.

9.3.3 Arrays

For the two-dimensional arrays, which are most common, how does the compiler handle *A*[*i*, *j*]? It depends upon the language specification about the array layout in the memory:

Row-major order: (most languages) Laid out as a sequence of consecutive rows. Rightmost subscript varies fastest, like: *A*[1,1], *A*[1,2], *A*[1,3], *A*[2,1], *A*[2,2], *A*[2,3]

Column-major order: (Fortran) Laid out as a sequence of columns. Leftmost subscript varies fastest, like: A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Indirection vectors: (Java) Vector of pointers to pointers to … to vectors of elements. This layout takes much more space, trades indirection for arithmetic and is not amenable to analysis. This is also one of the methods used in scientific computations in C. (See, for example, "Numerical Recipes in C".)

Figure 9.4 shows various layouts of an array.

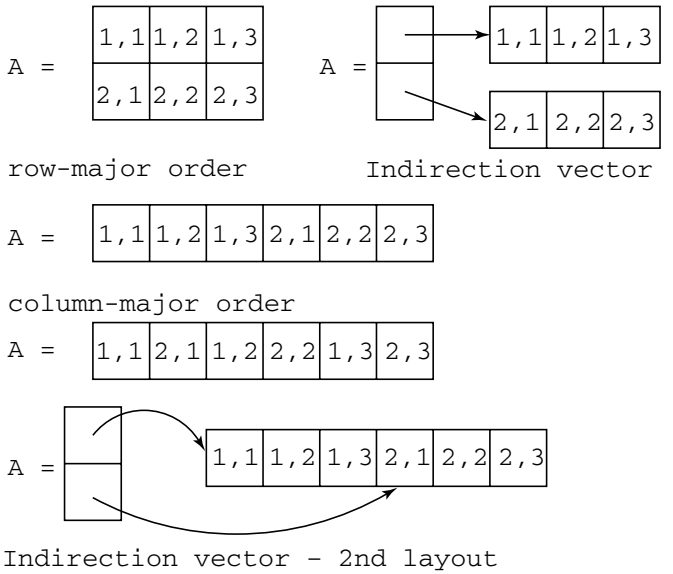


Fig. 9.4 Layout of an array in the memory. The indirection vector method has two variants, depending upon how individual rows are allocated space

Computing the array addresses: We illustrate the cases of two-dimensional arrays, A[rmax,cmax].

Row-major order: base(A) + (r*(cmax) + c)*sizeof(A[1])

Column-major order: base(A) + (c*(rmax) + r)*sizeof(A[1])

Indirection vectors: *(A[r])[c] where A[r] is a one-dimensional array reference, for example base(A) + r*sizeof(A[1])

Array references: To pass arrays as actual parameters to a function, a pointer to its first element and the size of each of the dimensions is given. In applications where a number of arrays are to be handled, it may be advisable to form a structure having these three values, which is traditionally known as *dope-vector*.

If the array being used in a function is a variable-sized array for example a local array dimensioned by actual parameters passed to the function, the array is to be allocated memory space at run-time.

Another aspect is array address calculations which is a major source of overhead in:

- Scientific applications make extensive use of arrays and array-like structures;
- Computational linear algebra, both dense and sparse;
- Non-scientific applications also use arrays – hash tables, adjacency matrices, tables, structures, etc.

The brighter side is that array calculations tend to iterate over arrays inside loops, and usually loops execute more often than code outside loops.

For example, consider the following code segment, where two conformal matrices are being handled.

```
for(j = 0; j < n; j++)
   a[i][j] = a[i][j] + b[i][j];
```

A naïve approach would be to perform the address calculation twice:

```
for(j = 0; j < n; j++){
   R1 = base(a) + (i * n + j) * (sizeof(A[1,1]))
   R2 = base(b) + (i * n + j) * (sizeof(A[1,1]))
   MEM(R1) = MEM(R1) + MEM(R2)
}
```

The inefficiency introduced is due to localized translation.

A more sophisticated way would be to move common calculations out of loop (see Chapter 10):

```
R1 = i * n * sizeof(A[1,1])
R2 = base(a) + R1
R3 = base(b) + R1
for(j = 0; j < n; j++){
   a = j * sizeof(A[1,1])
   R4 = R2 + a
   R5 = R3 + a
   MEM(R4) = MEM(R4) + MEM(R5)
}
```

An even more sophisticated method is to convert the multiplication to addition:

```
R1 = i * n * sizeof(A[1,1])
R2 = base(a) + R1
R3 = base(b) + R1
a  = sizeof(A[1,1])
for(j = 0; j < n; j++){
   R2 = R2 + a
   R3 = R3 + a
   MEM(R2) = MEM(R2) + MEM(R3)
}
```

Structures and Records

Structures and records have two complications:

- Each declared structure has a set of fields
  - Size and offset
  - Compute base + offset for field
  - Use size to choose load width and register width
- Structures and records can have dimensions
  - Arrays of structures
  - Fields that are arrays or arrays of structures
  - Use array address calculation techniques, as needed

Structures and records require compile-time support in the form of a table that maps field names to <offset,size> tuples.

Representing and Manipulating Strings

Character strings differ from scalars, arrays and structures in several ways. The basic unit is a character, whose typical sizes are 1- or 2-bytes. The target processor may or may not support character-size operations. Even then, the set of supported operations on strings is usually limited – assignment, length, concatenation and translation are usually available. On RISC processors, efficient string operations are complex.

Strings are represented usually by one of the two methods – explicit length field (as in PASCAL) and null termination (as in C). Strings' length may also be fixed and varying. Each method has its advantages and disadvantages.

9.4 Control Constructs

We now introduce a code generation scheme for statements:

```
S[if(e) s1 else s2]  =      E[e]
                            cmpl $1, %eax
                            jnz   lab1
                            S[s1;]
                            jmp   lab2
                      lab1: S[s2;]
                      lab2: ...
```

Here, `lab1` and `lab2` are new labels and

```
S[e;] = G[e]
```

For a WHILE-DO loop,

```
S[while(e) s] =  lab1: E[e]
                       cmpl $1, %eax
                       jnz   lab2
                       S[s]
                       jmp   lab1
                 lab2: ...
```

Here, `lab1` and `lab2` are new labels.

An alternative code generation rule for a WHILE-DO loop:

```
S[while(e) s] =         jmp lab2
                  lab1: S[s]
                  lab2: E[e]
                        cmpl $1, %eax
                        jz    lab1
```

The constructs of high-level languages can be implemented using sequences of machine instructions. It is important to select good target instructions for each source construct. Code Generation

schemes can be used to describe the mapping between the IR and target language. To get better code quality, we should:

- Make good use of the target's instructions, registers, etc.
- Use refined compilation schemes to deal with special cases.

9.5 Procedures and Function Calls

Functions are an important and widespread abstraction mechanism in many different programming languages. A function is a first step to reusable code. A function name carries its *returned* value. A procedure is a function that does not return any useful value via its name, and it is used for its side effects only. We now discuss the implementation of functions and the kind of data structures we require to implement them.

There are a wide range of different design choices:

- What are the types of arguments that can be passed as parameters or returned as results?
- Do we allow variable number of arguments?
- Do we allow recursion in functions?
- Do we allow nested functions?
- Can functions have side effects?

Various languages like FORTRAN, C, C++, Java, Python, Perl and Haskell would have different answers to the questions above.

There are several issues in actual implementations of functions:

- How does the function determine the correct return address on its exit?
- How are call parameters passed to functions?
- How are function results returned?
- Where are the values of local variables stored?
- What happens to local variables when the function returns?

The answers to these questions determine the *calling conventions* for a particular implementation.

Suppose we start with a naïve calling conventions and assign fixed variables for function parameters, locals and results. That will have many limitations.

Return address: A function may be called from many different points in a single program that is in fact the idea behind a function. The code for the function must be able to find the appropriate point in the program to *return* to after a function call is complete. Also if there are nested calls to a number of functions, the last one called should be the first to return. This indicates a LIFO or stack structure. Thus using `call` and `ret` instructions in the instruction set of a typical processor should be used. This also allows nested and recursive calls.

Return value: It is awkward and inefficient to access the return value of a function through a global variable. It is much better to use a register. However, the return value may not fit within a register, for example `eax` is 32-bits, and cannot return a `double`. Also, we have to take care that return value from repeated calls do not overwrite the register.

Finding parameters: If we use fixed variables to pass function parameters, we will get strange results if there are recursive function calls or several calls to the same function. Consider `factorial(n)` written recursively. There, intuitively, we expect a new, but temporary, variable `n` each time we enter the function. Parameters can be passed to a function by pushing values onto the stack, because

then the LIFO behaviour will automatically be achieved. For example, a call `f(x, y, z)` may be implemented by

```
pushl    x
pushl    y
pushl    z              // give parameters
call     f              // call function
addl     $12, %esp      // discard parameters
```

Note that we remove and discard the pushed values on return from the function. The general principle here is: the calling routine pushed the parameters, so it must remove them. It also means that the called function should not disturb them on the stack.

Accessing parameters: We have to access the values of function call parameters within the body of the function, without disturbing them on the stack. We should use *indexed* addressing, relative to the stack pointer, as shown in Fig. 9.5. For example, GAS code `4(%esp)` refers to the contents of the memory at address `esp + 4`. We replace uses of the parameter name `x` with `12(%esp)`, etc. However, this mapping has a problem. It can change during evaluation within the function, as the stack pointer moves due to other activity within the function. For example, we may *spill* some register value on the stack (see Section 9.8.1), while saving some intermediate value.

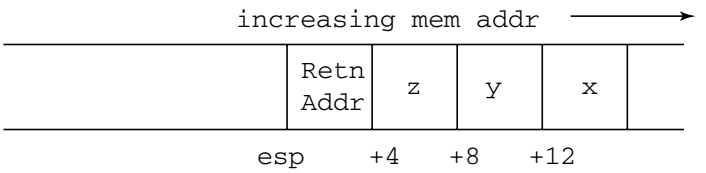


Fig. 9.5 Call parameters' access

Most of the modern processors provide a special register, called a *frame pointer*, `fp`, or a *base pointer*, `bp`, to establish the lower limit or the base of the stack-frame used by a function. On entry to a function, this register is set from the stack pointer, but once set, does not change for the duration of the function invocation.

The standard calling convention on the x86 processors uses the base pointer register, `ebp`, and it is used to access the call parameters. For each function or procedure call, we set up an Activation Record (AR) or stack-frame, as shown in Fig. 9.6. The arrangement now can be summarized as:

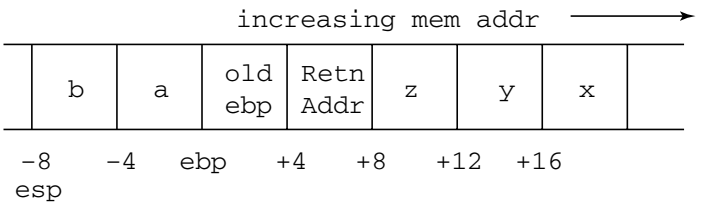


Fig. 9.6 A standard x86 stack-frame. Call parameters are accessible `8(%ebp)` onwards. Local variables are stored and accessible `-4(%ebp)` downwards

Call parameters: x, y and z are the function's call arguments. We can refer to z as `8(%ebp)`, y as `12(%ebp)`, etc.

Return address: It is at `4(%ebp)`, but it will be automatically used when the `ret` instruction is executed at the exit of the function.

Old base pointer: It is the base pointer for the calling function. This will provide a link to calling functions' stack-frame, in case it is also statically (lexically) enclosing.

Local variables: a, b are the local variables of the function. They are accessible, for example a as `-4(%ebp)`, b as `-8(%ebp)`.

Stack-frame: The caller starts by `pushing` the arguments, then it executes a `call` instruction, which `pushes` the return address. The called function saves the old base pointer, and sets a new value *equal to current stack pointer;* this establishes the base of the stack-frame. Then the function decreases the stack pointer to reserve space for the local variables.

9.5.1 Function Prologue and Epilogue

The code that builds the part of the stack-frame for which it is responsible, at the beginning of a function, is called the *prologue*. On the entry to a function, the call parameters and return address have already been pushed onto the stack by the caller. The function needs to:

```
pushl  %ebp          // save old base pointer
movl   %esp, %ebp    // and set current value
```

If there are local variables which require N bytes of storage, then the function needs to reserve space for them:

```
subl   $N, %esp      // allocate space for locals
```

That ends the function prologue.

When the function terminates, we must dismantle the stack-frame and return the machine to the state it was in just before the call to this function. The code to do this is called the function epilogue, which is basically running the previous procedure in reverse:

```
movl   %ebp,%esp     // discard locals and temps
popl   %ebp          // recover original base ptr
ret                  // return
```

On an x86 processor, the first two instructions can be replaced by the more efficient, but otherwise equivalent special instruction `leave`.

Once the called function returns to the caller, the result of the function is possibly in `eax`, but the parameters are still on the stack. They need to be removed. We restore the stack pointer to its original value by adding on the number of bytes, say M, used by the parameters:

```
addl $M, %esp
```

If no parameters were passed, then this step can be omitted.

9.5.2 Saving Registers

Consider what happens when a routine calls a function in the middle of an expression when there are values in registers that it needs to preserve. Unless the routine was written with this in mind, the

called function might overwrite those registers with values needed. Some part of the code has to take responsibility for preserving register contents, for example by saving them on the stack. There can be two approaches:

The caller: Knows which registers have values that must be preserved, but does not know which the callee will actually use.

The callee: The same situation as for the caller, but reversed.

Neither one has enough information to guarantee that it can do a good job. Inter-procedural register allocation can solve these problems if the caller and callee are in the same compilation unit.

9.5.3 A Test for C Linkage

We are planning to use the standard C linkage in our implementation of the code generator for `miniC` language discussed in Chapter 12. Also, as mentioned in Section 9.6, we do not plan to use standard C library for system services. It is required that we check that our plan for the stack-frame layout works with a C program. We wrote the following programs – first a function in assembly language as per suggested function calling conventions, and second a C program to invoke that function. Test function `myfunc1` in assembly:

```
.section .text
.globl    myfunc1
         .type myfunc1, @function
myfunc1:
         pushl  %ebp
         movl   %esp, %ebp
         movl   8(%ebp), %eax
         addl   $100, %eax
         movl   %ebp, %esp
         popl   %ebp
         ret
```

Test program `testclinkc.c` in C:

```
extern int myfunc1(int i);
int main(){
  int i = 5;
  printf("%d %d\n", i, myfunc1(i));
}
```

The C program was compiled with `gcc`, the assembly program was assembled with `as` and the two objects were linked. When `testclinkc` was executed, we got the correct answer “5 105”. We used this compatibility to test all the code generated by the `miniC` compiler.

We saw that a wide variety of design decisions are involved in the implementation of a function calling mechanism, with varying degrees of flexibility. Return addresses, function call parameters and local variables can be allocated storage on the hardware run-time stack. The most important data structure here is the Activation Record, also called a Stack Frame. Function parameters and local variables can be accessed from the current AR using either the stack pointer (unreliable) or a base pointer (preferred).

9.6 The Target Operating Environment

We have discussed at length the run-time environment in Chapter 7. In all subsequent discussions, we shall assume Linux Operating System (Kernel version 2.4.x or 2.6.x as mentioned) working on an x86 processor. We shall use GNU GCC for developing the software, GNU `gdb` for debugging support and use GNU `binutils` for various utilities to be used during program development.

We do not plan to use the standard C libraries, except for testing portions of compiler code. Wherever required Linux System Calls interface will be used for requesting system services like read and write to files, time of the day, etc.

9.6.1 Memory Management

Linux uses virtual memory system, details of which are given in Appendix C. We have also discussed the run-time memory management in Chapter 7.

9.6.2 CPU Register Usage

Although in x86 series machines almost any register can be used anywhere, there are some specializations. We take the following decisions after studying this aspect of the instruction set of x86:

1. Use `%eax` for a very short range of code because it is used in many short versions of instructions.
2. Use `%ecx` for shift and loop counts, addressing, saved addresses.
3. Use `%edx` for multiply, divide, short term only.
4. Use `%ebx` auxiliary medium term, also short-term array addressing.
5. Use `%ebp` reserved for forming stack-frame base pointer and stack addressing base.
6. Use `%esp` reserved for stack management, stack-based addressing.
7. Use `%esi`, `%edi` if not handling array indexing, then use as auxiliary; if array addressing then short-term address generation, possibly with `%ebx`.
8. Array address generation: ({`%ebx`, `%esi`, `%edi`}) + displ or (`%ebx`) + ({`%ebx`, `%esi`, `%edi`}) + displ or (`%ebx`) + ({`%ebx`, `%esi`, `%edi`})*{1,2,4,8} + displ.
9. Stack-frame-based addressing (`%ebp`) + ({`%ebx`, `%esi`, `%edi`}) + displ or (`%ebp`) + ({`%ebx`, `%esi`, `%edi`})*{1,2,4,8} + displ.
10. Up to 4-component addressing: (base) + (index)*scale + displ.
11. Define assembly macros only for those sequences which have no chance of being reduced by optimization, e.g. input output SysCalls.
12. `%eds`, `%ess`, `%ecs`, etc. are not used at all, neither should they be touched.

9.6.3 Activation Record (AR)

We have already discussed the need and use of activation records in Chapter 7. There we discussed AR in general, but here we shall be more specific and give a design of AR for the project on a compiler for a `miniC` language, which we shall discuss in Chapter 12.

To remind you, an AR is created on the Run-Time stack for each invocation of any function. An AR contains:

- return address (IP or PC);
- call arguments;
- saved BP (`%ebp`), SP (`%esp`);

- automatic variables;
- temporary locations, as decided by the compiler.

Figure 9.7 shows a typical Activation Record, according to our design. In deciding upon this layout of the AR, we had the following goals in mind:

1. It should use the `x86` subroutine calling idioms.
2. It should be based on "caller cleans the stack" convention, so that the possibility of variable number of arguments can be used if and when needed. It will also help make it compatible with the GCC C language calling conventions.
3. It should be compatible with the GCC C language calling conventions. This means that a set of functions compiled using the `miniC` compiler will be callable in standard C programs (and hopefully, vice-versa). This compatibility will be very convenient while debugging the generated code, as the functions written in `miniC` can be tested using test programs written in C.
4. It should be easy to generate the function definition and invocation codes.

Figure 9.7 shows the operation on the run-time stack, which results in creating an AR, during a function call. Figure 9.8 shows a mnemonic aid to remember the positions of various values in an AR.

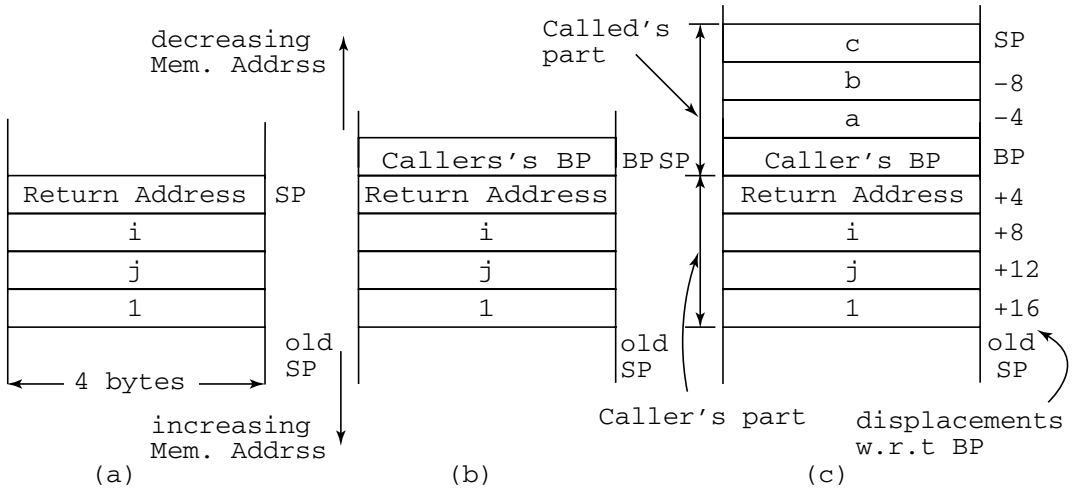


Fig. 9.7 Creation of an AR during a function call, (a) caller pushes the call arguments, (b) just after entry to the called function, (c) while the called function is executing

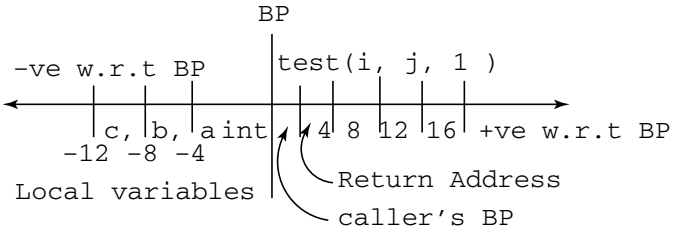


Fig. 9.8 Positions of various parameters and values within an AR corresponding to a function call `test(i,j,1)`, where the function `test()` itself has three local variables

9.7 Code Optimization

An *optimizing* compiler tries to generate the target code which a good assembly language programmer, intimately knowledgeable about the target processor, would have written. In this process, the optimizer replaces some naïve programming constructs used by the High Level language programmer, by more efficient target code sequences, at the same time maintaining the semantics of the source code. Then what is the real issue? The point is: an optimizing compiler will also provide a similar optimization across the different processor architectures. We want *portability* and *optimized code* at the same time . If a good assembly language programmer had to write a similar code for different processor architectures, he/she will have to put in considerable effort in learning the assembly languages, its idioms and applying them to the task at hand.

Two goals of optimization are:

Execution time reduction: Try to reduce the execution time of the compiled code.

Memory space reduction: Try to reduce the memory space requirement of the compiled code.

There are two levels at which optimization is generally attempted:

Machine-independent optimization: Consider the following code segment:

```
for(i = 0; i < 10000; i++){
    c = 5;
    a = d[i] + c;
}
```

In this code segment, the statement `c = 5;` does something which is independent of the progress of the loop and its value is not changed during the loop. It can be executed only once outside the loop and still the semantics of the code segment is preserved while execution time is reduced.

```
c = 5;
for(i = 0; i < 10000; i++){
    a = d[i] + c;
}
```

This kind of code adjustment and reduction of execution time and/or memory space is valid for any processor, as they are essentially independent of the processor characteristics. They are generally attempted *before* Code Generation takes place. However, we shall discuss them in Chapter 10. Some of the readers may feel that "I would have done the same thing myself if I had written the code". Yes, most of the machine-independent optimization counteracts carelessness, laziness, lack of knowledge or available time to think on part of the programmer. However, the second type of optimization takes actions which are not expected from an average high level language programmer.

Machine-dependent optimization: For example, store some of the intermediate results in the computation in the processor register, rather than in the memory. The processor registers typically provide 80 times faster access compared to the main memory and thus it will lead to a more efficient program if most of the computation can be done in the processor registers. On the other hand, processor registers are costliest of memories. Some sophisticated hardware arrangements are made to deal with the large gap in the speed and cost of registers and memory. High-speed *cache*' memories, generally known as Level-1 and Level-2 cache', are introduced between the CPU and the main memory. Refer to Fig. 9.9, where we have shown the memory hierarchy as obtained in a modern general-purpose

computer. We can safely say that a major part of the machine-dependent optimization effort is related to the issue of how to get maximum performance out of the memory hierarchy, keeping in mind the way the whole arrangement works.

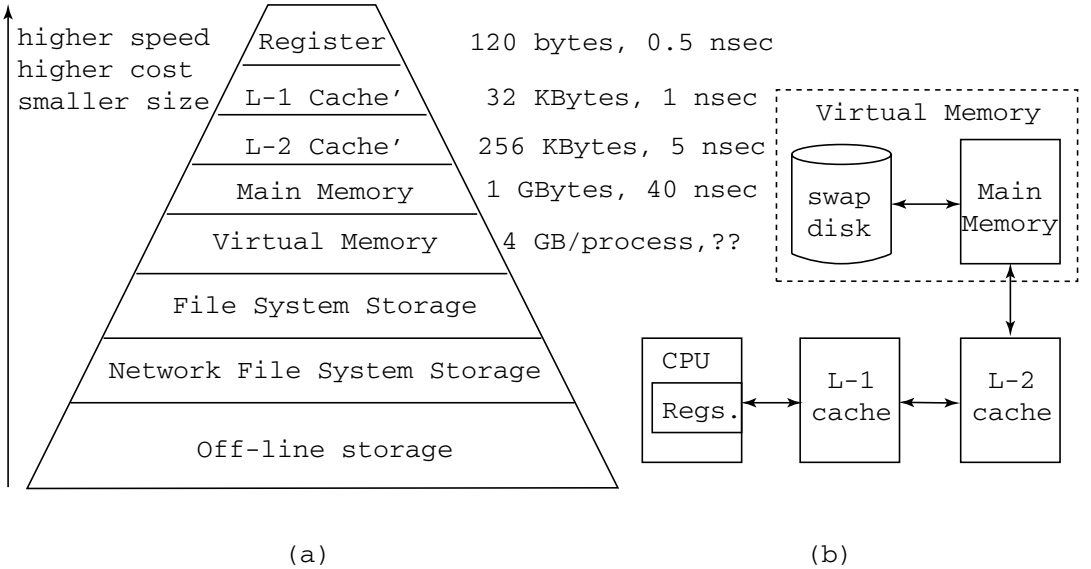


Fig. 9.9 (a) Memory hierarchy. The typical sizes and access speeds are noted on the side of each type of storage of our interest in code optimization. (b) The information flow in the top layers of the memory hierarchy. Registers store immediately needed (within a few instructions) information, L-1 cache stores about a kilobyte of immediately needed data and instructions of the most active processes, L-2 cache provides very roughly one active page (4 kb) per active process

The behaviour of the memory hierarchy can be roughly summarized as follows:

- Depending upon the instruction being executed, the CPU tries to access data in register/s or a memory address. If a register is being accessed, the data is available very quickly, say 0.5 nsec. The instruction can be completed at full processor speed.
- If a memory access is done, then if the address falls within the range of addresses whose data is held within L-1 cache, it is also available very fast, e.g. 1 nsec. Due to the small size of L-1 cache', only limited data can be held there. If the data requested is not in the L-1 cache', then the cache' has to be loaded from L-2 cache'. There are two situations in which this can arise: first, if in the currently executing program there is a *jump* instruction, whose jump target falls outside the range held in L-1 cache', second, the CPU starts another process – due to multi-tasking or an interrupt. The problem is that L-1 cache' just does not load only the required data, it loads data from L-2 cache' in chunks of say 128-bytes. This takes time, known as the *latency* of L-1 cache'. A very important aspect of all semiconductor memories, including cache', is latency. All memory storages have a certain latency, which means that a certain number of clock cycles must pass between two operations, like reads. L-1 cache' has lesser latency than L-2, typically 2 clock cycles or about 1 nsec, which is why it is so efficient.

- The L-2 cache' is normally much bigger, such as 256, 512 or 1024 Kb. The purpose of the L-2 cache' is to constantly read in slightly larger quantities of data from the main memory, so that the information is available to the L-1 cache'. The latency of L-2 cache' is 6 clock cycles or 3 nsec. Note that when L-1 is to be reloaded, it is the latency of L-2 which is applicable.
- The comments about reloading of the L-1 cache' from L-2 also apply to loading of L-2 cache' from the main memory and the same issues of code replacement apply, but now with 3 times the penalty time.
- The reloading of L-2 cache' involves a memory read and will take about 40 nsec per 8-bytes (usual memory data bus width).

In summary, from machine-dependent code optimization viewpoint, use as much of registers as possible, avoid jumps – and especially conditional jumps as much as possible, avoid memory access wherever you can and, to utilize the speed-up made possible by pipelined instruction execution, reorder instructions.

In the remaining part of this chapter and in the next chapter, we shall discuss various strategies for optimization. We shall label each method by the following labels to denote their basic characteristics:

MD: Machine-dependent optimization method
MI: Machine-independent optimization method
TR: The method may give reduction of execution time
SR: The method may lead to reduction of memory space requirements

There are three categories of optimization:

Front-end optimization: Where the optimization actions are generally implemented in the front-end of the compiler, but some of them may be done during code generation [MD, MI, TR, SR]. They are discussed in Chapter 10 in more detail. Replacing a simple function by in-line code is an example of this kind of optimization, which can be considered both MD and MI.

Peep-hole (Linear) optimization: Actions are done on the IR, but some of them are done on the raw target code also [MD, MI, TR, SR].

Structural optimization: Examines the total program or a substantial part of it. Usually done at IR level, but takes a more global view of the code [MI, TR, SR].

9.8 Machine-dependent Optimization

The naïve code generator that we have demonstrated several times does generate code which will execute (after assembly and linker steps) but the *code quality* is rather poor, in the following respect:

- Program size is big because the generated instruction sequences are quite long.
- Program execution time is long because there are redundant instructions and/or memory transfers.
- There is an inadequate use of CPU registers.
- The code generator does not make good use of the facilities of the machine, in the form of power of the instruction set.

An example: with the *E* compilation scheme, the statement:

```
total = total + count * count;
```

compiles to:

```
movl   total, %eax
pushl  %eax
movl   count, %eax
pushl  %eax
movl   count, %eax
popl   %ebx
imull  %ebx,  %eax
popl   %ebx
add    %ebx,  %eax
movl   %eax,  total
```

This translation has 10 instructions, 8 of which are accessing memory. Some of the memory accesses are unavoidable, but some of them seem avoidable if we use registers.

9.8.1 Register Allocation

Making Better Use of Registers

Up to now we have used only two registers in the generated code. Now let us take another extreme and imagine that we have as many registers as we need. We call them *r*(0), *r*(1), *r*(2), *r*(3), *r*(4), *r*(5), …. Can we generate a better code with all these registers available?

One trick is to store our stack of intermediate results in these registers and instead of using a stack pointer at run-time, we use our own stack pointer, `sp`. The compiler can keep track of where this stack pointer `sp` will be during compile-time. This arrangement leads us to another compilation scheme to generate code using our stack of registers.

`Er[e](sp)` generates code that will evaluate the expression `e` and leave the result in `r(sp)`, without changing any of the registers below `r(sp)`. For example,

```
Er[e1 + e2](sp) = Er[e1](sp)
                  Er[e2](sp + 1)
                  addl r(sp + 1), r(sp)
```

The scheme of arranging for intermediate values to be held in registers rather than memory locations is known as *register allocation*. That is how the name Er for the new compilation scheme has come about – *E* for expressions, *r* for register allocation. There are many different algorithms for register allocation. The approach we use here is one of the simplest, but not likely to be used in real compilers.

Some more examples are:

```
Er[n](sp)         =     movl $n, r(sp)
Er[v](sp)         =     movl  v, r(sp)
Er[el && e2](sp)  =     Er[e1](sp)
                        cmpl $1, r(sp)
                        jnz lab1
                        Er[e2](sp)
                  lab1: ...

Er[v = e](sp)     =     Er[e](sp)
                        movl r(sp), v
```

Compiling Statements

The rules for compiling statements need to be modified for the register-based code. Note that the intermediate values are used only while evaluating an expression, and the stack is expected to return to its original state at the end of evaluation of the expression. We can thus assume the stack is empty when we need the value of an expression:

```
Sr[e]            =      Er[e](0)
Sr[while(e) s] =        jmp lab2
                  lab1: Sr[s]
                  lab2: Er[e](0)
                        cmpl $1, r(0)
                        jz lab1
```

Taking the example statement `total = total + count * count:`

```
Sr[total = total + count * count]
         = Er[total = total + count * count](0)

         = Er[total + count * count](0)
           movl r(0), total

         = Er[total](0)
           Er[count * count](1)
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           Er[count * count](1)
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           Er[count](1)
           Er[count](2)
           imull r(2), r(1)
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           movl count, r(1)
           Er[count](2)
           imull r(2), r(1)
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           movl count, r(1)
           movl count, r(2)
           imull r(2), r(1)
```

```
          addl r(1), r(0)
          movl r(0), total

        = movl total, %eax
          movl count, %ebx
          movl count, %ecx
          imull %ecx, %ebx
          addl %ebx, %eax
          movl %eax, total
```

In the last step, we replace the assumed registers with real registers of the processor. Now we are making better use of registers. Any code to implement this particular statement would have to do two loads, one add, one multiply and one store. So now we are just one instruction away from optimal.

To make this compilation scheme work properly, we need to do a stack simulation, by adding a compile-time parameter to tell us what the stack size will be at the corresponding point at run-time.

Suppose we also try to track the run-time contents of the stack at compile-time. We shall write a stack in the form [*e*0, …, *en*] where *e*0, …, *en* are expressions, or a special symbol '?' which represents unknown.

Going back to our example code segment:

```
Sc[total = total + count * count]
         = Ec[total = total + count * count] []

         = Ec[total + count * count][]
           movl r(0), total

         = Ec[total] []
           Ec[count * count] []
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           Ec[count * count] [total]
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           Ec[count][total]
           Ec[count][total]
           imull r(2), r(1)
           addl r(1), r(0)
           movl r(0), total

         = movl total, r(0)
           movl count, r(1)
          Ec[count] [total, count]  The stack description here shows that
                                    count has already been loaded into
                                    register r(1) so we do not need to

           movl r(0), total
                                    load it again now
           imull r(2), r(1)
           addl r(1), r(0)
           movl r(0), total
```

Now we note that:

```
Ec[v][e0, ..., en]
              = movl r(i), r(n + 1)   if ei = v
                movl v, r(n + 1)      otherwise
```

This gives us:

```
Sc[total = total + count * count]
  = movl total, r(0)
    movl count, r(1)
    movl r(1), r(2)
    imull r(2), r(1)
    addl r(1), r(0)
    movl r(0), total
```

This is one more instruction than what we needed, but we have eliminated the duplicated memory access. The redundant instruction too can be eliminated using an optimization method known as *copy propagation*.

Some instruction sequences can invalidate the cache entries. For example, consider:

```
(total + (total = 2) + total)
```

Naïvely applying the previous compilation scheme would give us a code like the following:

```
movl total, r(0)
movl $2, r(1)
movl r(1), total    Total has been changed, so we cannot use r(0) as a shortcut
                    to total anymore. And, in fact, we are overwriting r(0)
addl r(1), r(0)     In fact, we are overwriting r(0) here with another value
                    anyway.
movl r(0), r(1)
addl r(1), r(0)     So using the value in r(0) here is definitely a mistake!
```

To avoid such problems, we have to keep the cache accurate, by tracking the cache more carefully.

```
movl total, r(0)[total]
movl $2, r(1)[total, 2]
movl r(1), total[?, total = 2]    The cache has two different symbolic
                                  representations of the value in r(1) at
                                  this point
addl r(1), r(0)[?, total = 2]     But the original value for total has been
                                  overwritten.
movl r(1), r(1)
addl r(1), r(0)[?]
```

This shows that we can determine considerable information about the run-time behaviour of a program by using symbolic data at compile-time. One has to be careful about subtle pitfalls and oversights. This is an area where formal methods can help.

In the real world, no processor has infinitely many registers built into it. An x86 series of processor has:

Accumulator `eax`

Base `ebx`

Count `ecx`

Data `edx` – cannot be used as a general register, used in `mult` and `div` instruction as extension of `eax`.

Source index `esi`

Destination index `esi`

Base pointer `ebp` – cannot be used as a general register, used as stack-frame base pointer.

Stack pointer `esp` – cannot be used as a general register, used as run-time stack pointer.

Instruction pointer `eip` – cannot be used as a general register, hardware instruction pointer.

Flags `eflag` – cannot be used as a general register, hardware flags.

After we have eliminated special purpose registers, an x86 has only five general-purpose registers available.

Register Pressure

The number of registers that we need for a given computation is sometimes described as *register pressure*. High register pressure occurs most frequently when we are compiling for processors with very few registers. We can deal with register pressure by using registers more carefully and using register *spilling* when all else fails.

To illustrate the first solution, i.e. "Use Registers Carefully", let us take our previous example;

```
S[total = total + count * count]
   = movl   total,  r(0)            use register r(0) to hold the value
     movl   count,  r(1)            of total we do not actually need the
     imull  r(1),   r(1)            value of total until after we have
     addl   r(1),   r(0)            calculated count * count. We may delay
     movl   r(0),   total           the load of total
```

We rewrite the code as:

```
S[total = total + count * count]
   = movl   count,  r(0)
     imull  r(0),   r(0)
     movl   total,  r(1)
     addl   r(1),   r(0)
     movl   r(0),   total
```

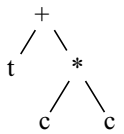
This has not changed the number of registers that we need, but it has shown a possibility for better code. Instead of loading total into r(1), we can add it directly to r(0).

```
S[total = total + count * count]
   = movl   count,  r(0)
     imull  r(0),   r(0)
     addl   total,  r(0)
     movl   r(0),   total
```

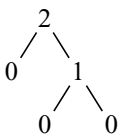
Now we are using only one register and also fewer instructions. These improvements would be particularly important if we were dealing with a subexpression of some larger statements where more registers were needed.

Minimizing Depth

It can be proved that we use fewest registers if we evaluate the subexpression with greatest depth first. The syntax tree for our example is:



and the corresponding depth values are:



Note that application of the above rule will force evaluation of the RHS subtree, `c * c` first and then addition to `t`. If instead of addition we had subtraction and we evaluate the arguments of a subtraction in the reverse order, then we will have to use an `xchgl` instruction before `subl`. If we evaluate the arguments of an addition in the reverse order, then an `addl` by itself will be acceptable, as addition operation is commutative.

While applying the above rule we have to be careful about the side effects. If the arguments of an operator might have side effects, then we may not be able to change the order. For example, suppose that some function `f(x)` prints the value of `x` on the console. If we are generating code for the expression `f(3) + (1 + f(2))`, value 2 will be printed before 3.

Some languages specify an explicit order of evaluation. Others leave the choice unspecified to give the implementer more flexibility.

Register Spilling

The term register spilling refers to the process of moving temporary values from registers into the run-time stack. Conceptually, we use our finite set of registers, together with the memory stack, to simulate our infinite family of registers *r*(*i*):

Logical:	*r*(0)	*r*(1)	*r*(2)	*r*(3)	*r*(4)	*r*(5)	*r*(6)	…
Physical:	eax	ebx	ecx	esi	edi	eax	ebx	…

In practice, we modify our compilation schemes to use spilling to reserve space when they need a new register. For example,

```
Er[e1 + e2](sp) = Er[e1](sp)
                  spill(sp + 1)
                  Er[e2](sp + 1)
                  addl r(sp + 1),r(sp)
                  unspill(sp + 1)
```

where

```
spill(sp)     = pushl r(sp), if sp >= 5,
              = null,        otherwise
unspill(sp)   = popl r(sp),  if sp >= 5,
              = null,        otherwise
```

With only two registers, our example code requires:

```
S[total = total + count * count]
        = movl   total, r(0)
          movl   count, r(1)
          pushl  r(2)
          movl   count, r(2)
          imull  r(2),  r(1)
          popl   r(2)
          addl   r(1),  r(0)
          movl   r(0),  total
```

If we had just two registers, and had not made the other improvements, this is the code we would end up with. Note that $r(0) = r(2) =$ eax, and $r(1) =$ ebx.

The moral is: use registers when we can, and use spilling when we have to.

Making Use of Context

To get better code, we add information about the context in which the expression appears to our compilation scheme. Without context information, a compilation scheme must work in all places where that expression could appear. With context information, the compilation scheme needs only to produce code that works in the specified context.

Recurring Patterns

Recall the compilation rules for logical-AND and while-loop:

```
Er[e1 && e2](sp) =      Er[e1](sp)       |
                        cmpl $1, r(sp)   |*
                        jnz lab1         |
                        Er[e2](sp)
                  lab1: ...
S[while(e) s]    =      jmp lab2
                  lab1: S[s]
                  lab2:
                        Er[e](0)         |
                        cmpl $1, r(0)    |*
                        jz lab1          |
```

The blocks of code marked '*' on the RHS use the result of a Boolean expression to make a jump; they do not actually need the result itself. We give the two patterns a name

```
branchTrue[e](sp,label)  = Er[e](sp)
                           cmpl $1, r(sp)
                           jz label
```

```
branchFalse[e](sp,label) = Er[e](sp)
                           cmpl $1, r(sp)
                           jnz label
```

Note that we have added the label, an extra context, as an additional argument. We have now the new schemes by abstracting details from the compilation schemes. Now we can rewrite the compilation schemes to utilize them:

```
Er[e1 && e2](sp) =       branchFalse[e1](sp,label)
                         Er[e2](sp)
                 label:  ...
S[while(e) s]    =       jmp label2
                 label1: S[s]
                 label2: branchTrue[e](0,label1)
```

We can produce special compilation rules for some conditional jump constructs:

```
branchTrue[true](sp,label)  = jmp label

branchTrue[false](sp,label) = null

branchTrue [e1 || e2](sp,label) = branchTrue[e1](sp,label)
                                  branchTrue[e2](sp,label)

branchTrue [el && e2](sp,label) = branchFalse[e1](sp,label1)
                                  branchTrue [e2](sp,label)
                          label1:
```

We do get benefits from this improved code. For example, with the original compilation schemes, we get:

```
S[while(el && e2) s]   =     jmp label2
                     label1: S[s]
                     label2: Er[e1](0)
                             cmpl $1, r(0)
                             jnz label3
                             Er[e2](0)
                             cmpl $1, r(0)
                             jz label1
                     label3:
```

With the modified compilation schemes, we get:

```
S[while(el && e2) s]   =     jmp label2
                     label1: Srs[s]
                     label2: branchFalse[e1](0,label3)
                             branchTrue[e2](0,label1)
                     label3:
```

If either `e1` or `e2` fits the special cases that we have defined for `branchTrue` or `branchFalse`, then we will get even better code.

There are a few more compilation schemes which are of interest. The increment operator, ++, in Java, C and C++ can be implemented using:

```
var++ = movl var,%eax        ++var = incl var
        incl var                     movl var,%eax
```

If `var++` or `++var` appears in an expression, then the final value in eax is important. If either appears as a statement, then the final value is not important, and we can compile either as just `incl var`. We can distinguish the different contexts by matching for ++ expressions both in the *S* and the *E* schemes.

Java, C and C++ provide two ways of escaping from a loop:

```
while(...){          while(...){
    ...                    ...
  }                      cont:
                     }
                     past:
```

A `break` statement in the left loop is equivalent to `goto past` in the right one. A `continue` statement in the left loop is equivalent to `goto cont` in the right one. The innermost `cont` and `past` labels must be passed as context parameters to the `S` scheme, in order to support breaking and continuation of loops.

We have seen that with careful selection of compilation schemes, we can produce good quality assembly language code, directly from the verified representation of a program. Register allocation makes a big difference in performance and code size. Register spilling is used to cope with limited numbers of registers. Instruction scheduling can make a difference to execution time on modern machines. The more information that we have about the context of a program segment, the better is the code that we can generate for it.

9.8.2 Instruction Rescheduling: Use of Parallelism in Instruction Execution

An important aspect of modern processors' architecture which concerns machine-dependent optimization is *Instruction Pipelining*. This is a kind of parallel operation within a single processor. However, to derive maximum possible benefits from the hardware scheme, the instructions in the target code are to be *rescheduled*, i.e. their order of execution is changed, at the same time taking care to preserve the semantics of the original code.

Instruction Pipelining

The processor has several steps (or stages) in treating each instruction. The stages vary from one processor to another, but it can generally be broken down into following stages:

Fetch stage: In this stage, the processor fetches the instruction from the memory (or the cache).

Decode stage: In this stage, the processor tries to figure out the meaning of the instruction. Remember that the instruction is in the form of machine code.

Execute stage: After figuring out the meaning, the processor executes the instruction. Here, the processor also fetches any data in memory if needed. In that case, the execute stage is considered to be divided in two stages – *data fetch* and *execute*.

Write-back stage: If there are results need to be stored back to the memory, the processor dispatches them.

Each of these stages can actually be expanded into several stages. The old Pentium, for example expand the execution stage into two: one stage is to fetch any data from memory if needed, the other is to actually execute it. Also, in the Pentium, after the write-back stage, we have "error reporting" stage which is, obviously, to report errors if any. Even Pentium 4 has 20 stages.

Obviously, these stages are arranged into a single sequential pipeline so that the instruction enters from the first stage, sequentially to the end as shown in Fig. 9.10.

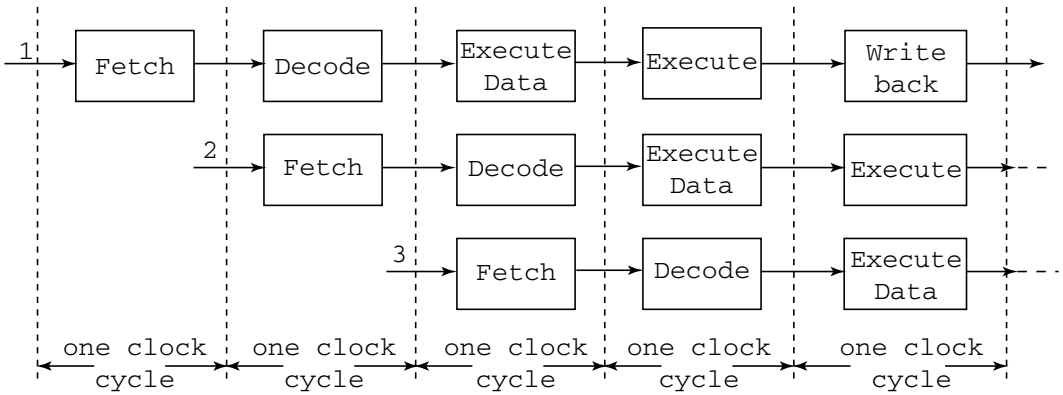


Fig. 9.10 Instruction execution pipeline. At a time more than one instruction are in various stages of their execution. The processor starts *Fetching* the next instruction as soon as the previous one enters its *Decode* stage. This is *anticipatory* fetching or pre-fetch. With such pre-fetch, the L-1 and L-2 cache' get filled up. A problem arises when a conditional jump, or even a simple jump, invalidates the pre-fetched instructions. Then the cache' has to be filled *ab initio*

Staging the instruction process has a great advantage. Suppose you have instruction 1 to n to be executed sequentially. At clock 1, instruction 1 enters the first stage. At clock 2, instruction 1 enters the second stage. Since at clock 2, the first stage is empty, we can feed in instruction 2. At clock 3, instruction 1 is at stage 3, and instruction 2 is at stage 2, so we can put in instruction 3 into stage 1, and so on.

The effect of all these is this: if we have four stages of processing, it is like processing 4 instructions at the same time (but in different stages). So the net effect is four times the "normal" performance. Thus pipelining certainly boosts up performance.

However, there is a catch: if we have conditional jumps then the performance boost may not be as expected. If instruction number 3 is being executed, thus in stage 3 in Fig. 9.10, the instructions in stages 1 and 2 must be cancelled if the *prediction* is wrong. What is *prediction?* The very fact that the jump instruction is in stage 3 means that the processor assumes that the jump, if at all taking place, will be such that the instructions which have entered stages 1 and 2 will in fact be executed next.

Today's computers have more than one pipeline. This is referred as multi-pipeline processor. The old Pentium had two pipelines, so it was like having two separate processors working simultaneously. If we have two pipeline, the processor can execute two instructions in parallel. If each pipeline has 5 stages, we effectively pump up the performance up to 10 times. Running two or more instructions in

parallel needs a precaution: These instructions must be independent of each other in order to be able to be executed in parallel. For example,

```
mov bx, ax
mov ex, bx
```

These instructions cannot be run in parallel, because the second instruction needs the outcome of the first instruction, i.e. the value of BX is determined by the result of the first instruction. Look at the next example:

```
mov bx, ax
mov ex, ax
```

This program segment can be run in parallel because now both of them only depend on AX, which is assumed already set previously. We know that both segments mean the same thing, but the second example is faster because it can run in parallel. Therefore, the instruction *reordering* can make the difference because of multi-pipelining.

The conclusion is: if you want to speed up your code, it is worth reordering the instructions so that many of them can be run in parallel.

9.9 Converting the 4-Tuple and RPN into Assembly Code

Although generating an assembly language code directly by the semantic phase is not only possible, but also comparatively easy, we have taken the path of the IR in anticipation of later possibility of machine independent optimization.

Considerable string matching and manipulation are involved in converting a 4-tuple code into an assembly code, so we chose to write the code generator in Perl script. We use an auxiliary file, called `codebase.mat` which contains the templates for the conversion, and its initial version for converting 4-tuple IR is given below:

```
DCL#  A2: .A1 0
LD#   movl A1, A2
=#    movl A1, A2
ADD#  addl A1, A2 nl movl A2, A3
SUB#  subl A2, A1 nl movl A1, A3
MUL#  imull A1, A2 nl movl A2, A3
ADDE# addl A1, A2
SUBE# subl A1, A2
MULE# imull A1, A2
BZ#   jz      labelA1
BR#   jmp     labelA1
LAB#  labelM:
RET#  movl A1, %eax nl movl %ebp, %esp nl popl %ebp nl ret nl _start:
DFN#  .globl A1 nl .type A1 @function nl A1: nl pushl %ebp nl movl %esp,
%ebp
CALL# call A1 nl addl $A3, %esp
ARG#  pushl A1
CARG# movl A1(%ebp), A2
```

Here, each line represents one template, whose format is:

```
<op-code># <code generation template>
```

*A*1, *A*2, etc. are tokens which will be replaced by appropriate quantities by the code generator script. The assembly language op-codes and register designations, if any, are reproduced without change or replacement. The string "nl" represents the next-line character to be inserted in the output. Generally, *A*1 will be replaced by the first argument in the 4-tuple, *A*2 by the second and *A*3 by the third argument.

We use a slightly different `codebase.ast` file for conversion of RPN IR.

9.9.1 Conversion of 4-Tuple to Assembly Code

The `codegen.pl` Perl script is small enough to be given here in its entirety.

```
#!/usr/bin/perl
# generate x86 assembly code from 4-tuple matrix created by miniC
open(MAT,"$ARGV[0]") or die;
@mat = <MAT>;
close(MAT);

open(CB,"codebase1.mat") or die;
@cb = <CB>;
close(CB);
%cba = ();
foreach $line (@cb){
  chomp($line);
  ($cbop, $code) = split /[\#]/, $line;
  $cba{$cbop> = $code;
}

$dcl = 0;
%syntabtype = ();
@regs = ("%eax", "%ebx", "%ecx", "%esi", "%edi");
foreach $r(@regs){$reg2t{$r} = 0;}
@t2reg = ();
$lastr = "";

# Collect and output all Variable declarations
print " .section .data\n";
foreach $entry (@mat){
  chomp($entry);
  ($mno, $op, $opr1, $opr2, $opr3) = split /[: ]+/,$entry;
  if($op =~ "DCL"){
    $dcl = 1;
    $code = $cba{$op};
    $code =~ s/A2/$opr2/;
    $code =~ s/A1/$opr1/;
    print "$code\n";
    $symtabtype{$opr2} = $opr1;
  }
```

```
}
prologue();

LOOP:
foreach $entry (@mat){
   chomp($entry);
   ($mno, $op, $opr1, $opr2, $opr3) = split /[: ]+/,$entry;
   next LOOP if($op =~ "DCL");
   $code = $cba{$op};
   if($symtabtype{$opr1}){$opr1 = "(".$opr1.")";}
   if($op !~ "B" && $op !~ "CARG" && $opr1 =~ /^[0-9]+/){$opr1 = "\$".$opr1;}
   $opr1 = regmap($opr1);
   $opr2 = regmap($opr2);
   $opr3 = regmap($opr3);
   $code =~ s/A1/$opr1/g;
   if($opr2 !~ "--"){$code =~ s/A2/$opr2/g;}
   if($opr3 !~ "--"){$code =~ s/A3/$opr3/g;}
   $code =~ s/nl/\n/g;
   if($op =~ "LAB"){$code =~ s/M/$mno/;}
   print "$code\n";
}
epilogue();

sub prologue {
  print " .text\n .globl _start\n"; # _start:\n
}

sub epilogue {
  print " movl  %eax, %ebx\n";
  print " movl  \$1, %eax\n";
  print " int   \$0x80\n";
}

# Replace temporaries Tn with x86 machine registers
sub regmap {
  $t = shift;
  if($t !~ /^T/){ return $t;}
  $t =~ /T([0-9]+)/; $tn = $1%5;
  if($t2reg[$tn]){$lastr = $t2reg[$tn]; return $lastr;}
  foreach $r (sort keys %reg2t){
    if($reg2t{$r} != 0){
       if($reg2t{$r} == $tn){$lastr = $r; return $r;}
    }
    else {$reg2t{$r} == $tn; $lastr = $r; return $r;}
  }
}
```

We give here just one example of the generated code. Consider the following `miniC` program with a function, in file `testf3.miniC`:

```
int: a, b, c
func myfunc() $1 + $2
a = 2
b = 3
c = b + 9
myfunc(a + 2,c - 3)
end
```

The corresponding 4-tuple IR, `testf3.MAT` generated was:

```
1:  DCL int a --
2:  DCL int b --
3:  DCL int c --
4:  DFN myfunc -- --
5:  CARG 8 T1 --
6:  CARG 12 T2 --
7:  ADD T1 T2 T3
8:  RET T3 -- --
9:  LD 2 T4 --
10: = T4 a --
11: LD 3 T5 --
12: = T5 b --
13: LD b T6 --
14: LD 9 T7 --
15: ADD T6 T7 T8
16: = T8 c --
17: LD a T9 --
18: LD 2 T10 --
19: ADD T9 T10 T11
20: ARG T11 -- --
21: LD c T12 --
22: LD 3 T13 --
23: SUB T12 T13 T14
24: ARG T14 -- --
25: CALL myfunc T15 8
```

When the above 4-tuple matrix was translated by `codegen.pl`, using the template file shown above, we get the following assembly language code:

```
.section .data
a: .int  0
b: .int  0
c: .int  0
  .text
  .globl _start
  .globl myfunc
    .type myfunc @function
myfunc:
pushl %ebp
movl %esp, %ebp
  movl 8(%ebp), %eax
```

```
  movl 12(%ebp), %ebx
addl %eax, %ebx
  movl %ebx, %ecx
  movl %ecx, %eax
movl %ebp, %esp
popl %ebp
ret
_start:
  movl $2, %edi
    movl %edi, a
  movl $3, %esi
    movl %esi, b
  movl (b), %eax
  movl $9, %ebx
addl %eax, %ebx
  movl %ebx, %ecx
    movl %ecx, c
  movl (a), %edi
  movl $2, %esi
addl %edi, %esi
  movl %esi, %eax
    pushl %eax
  movl (c), %ebx
  movl $3, %ecx
subl %ecx, %ebx
  movl %ebx, %edi
    pushl %edi
  call myfunc
addl $8, %esp
  movl %eax, %ebx
  movl $1, %eax
  int  $0x80
```

This assembly code was assembled using as `-o testf3.o testf3.s` and linked using the command `ld -o testf3 testf3.o`. It did execute properly and left correct value 13 in register `%ebx`.

9.9.2 Conversion of RPN to Assembly Code

The arrangement similar to 4-tuple conversion is done for RPN to the assembly conversion also, though some of the details of the `codebase.ast` and the conversion script are different. These differences arise mainly because of the fact that there are no *temporary* variables in an RPN IR. Also, the code generator will have to use a `code generator` stack to mimic the implied actions with respect to the operands. For example, RPN corresponding to infix expression `a * (b + c)` will be `a b c + *`. The three operands are pushed on the code generator stack. Then the + operation pops two operands from code generator stack, generates assembly instructions for loading operand values in registers and issues the ADD instruction. RPN expects that the result is pushed back on the stack. Thus, the code generator will have to create a *temporary*, push it on the code generator stack, then check if a register can be used in its place and generate code accordingly. Next when `*` operation is encountered, it pops two operands from the code generator stack (the TOS will be the temporary pushed by the + operation) and pushes

result back on the code generator stack as a second temporary, and so on. The register assignment to the temporaries is done in the same way as the code generator for 4-tuples.

We do not give here the complete Perl script for the code generator for RPN or the corresponding `codebase.ast` file. Rather only the assembly output obtained for a small `miniC` program shown below is given.

```
int : a, b, c, d
a = 3
b = 5
c = 7
d = a * (b + c)
end
```

The corresponding RPN output from the semantic phase is:

```
int a DCL int b DCL int c DCL int d DCL 3 a = 5 b = 7 c = a b c + * d =
```

and the corresponding assembly code generated is:

```
.section .data
a: .int 0
b: .int 0
c: .int 0
d: .int 0
  .text
  .globl_start
_start:
  movl $3, a
  movl $5, b
  movl $7, c
movl (b), %eax
  addl (c), %eax
pushl %eax
movl (a), %ebx
imull %eax, %ebx
pushl %ebx
  movl %ebx, d
  popl %ebx
  movl $1, %eax
  int  $0x80
```

When assembled by `as` and linked by `ld`, this code gave correct results. Note that the code generated in smaller compared to a naïve code generator for 4-tuples.

? EXERCISES

1. Develop a code generator for the Boolean expression calculator whose grammar G_b is given in Chapter 4.
2. Rewrite the code generator given in Section 9.9 in C, which is written in Perl language.

3. Develop your own code generator for mapping RPN into an assembly code. You may use C or Perl or any other language you are familiar with.
4. Many of the assembly language code sequences are recurring in a typical code generation step. Implement these common sequences as assembly language macros.
5. Keeping in view the Machine-dependent Optimization, how far the idea of defining and using assembly language macros for some frequently occurring code sequences is advisable?

FURTHER READING

1. Floating Point Operations in x86: Data movement: Load, Load Constant, and Store Arithmetic: Add, Subtract, Multiply, Divide, Square Root and Absolute Value Comparison: Instructions to send the result to the integer CPU so that it can branch Transcendental: Sine, Cosine, Log, and Exponentiation.
2. If you want to use floating point computations, you should be aware of their problems and how to avoid them; a good starting point is David Goldberg, "What Every Computer Scientist Should Know About Floating-point Arithmetic" (http://docs.sun.com/source/806-3568/ncg_goldberg.html), ACM Computing Surveys 23(1):5–48, March 1991.

GLOSSARY

Absolute address A specific address (identification number) permanently assigned to a storage location, a device or a register by the designer.
Address A unique identification number of some source or destination data or an instruction.
Addressing mode The manner in which the processor determines the operand addresses within an instruction.
Address space The number of locations in the memory which can be located by the addressing technique used by the processor.
Architecture The organizational structure of a processor or the computer system.
Arithmetic and logic unit (ALU) A digital circuit which performs arithmetic and logic operations on operands.
Assembler A program that translates assembly language program into a machine language program.
Auto-increment addressing mode The content of a register is used as memory address and then automatically incremented by 1, 2, 4 or 8: useful for an array access or string processing.
Barrel shifter A special type of shifting circuit which shifts the input operand by an amount determined by a second input, in one clock period: used to implement complex addressing modes.
Branch or jump An instruction which makes the program control take a path different from the normal linear path.
Bus cycle The period of time in which a device connected to a bus completes all necessary operations for a single-bus transfer.
Cache' memory A very high-speed, small, expensive, storage unit, which stores information (data or instruction) and which may be needed by the processor in near future.
Clock cycle A time period of the basic synchronizing mechanism within a processor.
Co-processor A specialized processor which executes specific functions independent of the main processor. Its presence in the system is expected to speed up computation.
Cycle stealing A technique used usually by direct memory access (DMA) for transferring data with the main memory, during which time the processor is prevented from using the memory busses.

Direct address An address that specifies the memory location of an operand in an instruction.
Effective address The final address of an operand or instruction, generated during an instruction execution, which goes to the memory system for the operand access.
Exception processing The processor state associated with interrupts, traps, tracing and other exceptional conditions.
Immediate address An address that is used as an operand value by an instruction.
Implied address An address not specified, but known implicitly by nature or op-code of an instruction.
Indexed address The effective address is generated by adding the content of a specified index register to the address contained within the instruction (usually known as a *displacement*).
Indirect address An address of a location where the address of an operand is stored.
Instruction set Specifies a processor's capability by listing all the instructions and details of its behaviour.
Linker Connects together programs which are compiled or assembled separately.
Loops A program control construct where a sequence of instructions is executed repeatedly till a specified termination condition is reached.
Machine language The language expressed in binary number system which a processor is able to follow without further translation.
Memory cycle time The minimum time which must elapse between two successive memory accesses.
Multitasking A property and technique of an operating system which allows more than one task to execute on a single processor concurrently.
Nested functions Definition of a function is embedded completely within the scope of another function, which is called lexical or static nesting. A function may call another function, and then it is called dynamic nesting.
Operand A datum or information item involved in the operation of an instruction and using which a result is computed.
Operating system A basic collection of programs which forms a layer of functionality above the bare hardware and provides resource (Processor, Memory, I/O devices, secondary storage) management and a virtual extended machine with more user-friendly interface.
Pipeline A technique by which a processor is able to initiate execution of more than one instruction at a time.
Pointer A storage location or a processor register which contains the address of some operand.
Register indirect address A register contains the address of an operand.
Relative address An address relative to a specified starting point.
Routine A group of instructions carrying out a specific activity.
Tracing A dynamic diagnostic technique in which a recording of the internal state of a processor permits later analysis of a program execution.
Virtual memory A hardware- and software-based memory management technique which permits the allocation of memory to individual programs much in excess of the physical main memory.

10
Code Optimization

What you will learn in this chapter

- What are the issues in Code Optimization?
- How are Basic Blocks found out?
- What is a Flow-graph?
- What is Value Numbering Scheme?
- What is Peep-hole optimization?
- What are the different methods of Peep-hole optimization?
- What is Structural Optimization?
- What are the different methods of Structural Optimization?
- How is Global Data-flow analysis done?
- What is Live variable analysis and how is it done?
- What are Super-optimizers?

Key Words

machine-independent optimization, flow-graph, value numbering, peep-hole optimization, structural optimization, global data-flow analysis, live variable analysis, super-optimizers

We have already discussed some preliminary ideas about Optimization in Chapter 9, Section 9.7. There the emphasis was on methods which can be used on the raw target code generated by the code generator. We now discuss methods which can be applied before the target code is generated, in other words the optimizing transformations are applied to the IR.

10.1 Basic Blocks

In simple English, a *basic block* is a sequence of statements whose first statement may have a *label*, the last statement is a Jump or Conditional Jump, and does not contain any other labels, Jumps or Conditional Jumps. That is, we can only enter at the beginning of a basic block and exit at the end. If you remember *Structured Programming* this was exactly the condition imposed on all the control constructs like SEQUENCE, IF-THEN-ELSE, WHILE-DO, FOR, REPEAT-UNTIL and SELECT-CASE, but here a lower level of control flow is being considered, in terms of Jump instructions only.

The concept of Basic Blocks is used to form what is known as *Flow-graphs*, wherein they appear as nodes. A flow-graph indicates which block can follow which other blocks. The idea of flow-graphs is

useful in machine-independent optimization. Also, a real-life compiler, the GNU gcc uses the concept of basic blocks in its optimization algorithms.

A little thought will tell you that for a given IR code, the basic blocks will be delimited by the targets of jumps, i.e. labels and end just before such labels or by jump commands. For example, consider the following 4-tuple IR for a IF-THEN-ELSE construct:

```
1:  DCL int a --
2:  DCL int b --
3:  DCL int c --
4:  DCL int d --
5:  LD 3 T1 --
6:  = T1 a --
7:  BZ 11 -- --
8:  LD 7 T2 --
9:  = T2 c --
10: BR 18 -- --
11: LAB 7 -- --
12: LD a T3 --
13: LD b T4 --
14: LD c T5 --
15: ADD T4 T5 T6
16: MUL T3 T6 T7
17: = T7 d --
18: LAB 10 -- --
```

The basic blocks, denoted by pairs (start, end), are $B1 = (1, 7)$, $B2 = (8, 10)$, $B3 = (11, 17)$, $B4 = (18, -)$.

10.1.1 Formal Algorithm to Delineate BBs

Algorithm 10.1.1 specifies a formal method of partitioning a program into Basic Blocks.

Algorithm 10.1.1: Basic Blocks

```
   /*Find all statements which start a Basic Block                    */
1  the first statement of the program, or;
2  any labelled statement target of a branch, or;
3  any statement following a branch;
   /*For each statement starting a BB, the BB consists of:            */
4  that statement, and;
5  all statements, but excluding, the start of the next BB, or;
6  end of the program.
```

For example, consider the following 4-tuple IR code for a WHILE with nested IF:

```
1:  DCL int a --
2:  DCL int b --
3:  DCL int c --
4:  DCL int d --
```

```
5:  LAB 5 -- --
6:  LD 3 T1 --
7:  = T1 a --
8:  BZ 24 -- --
9:  LD 5 T2 --
10: = T2 b --
11: BZ 19 -- --
12: LD a T3 --
13: LD b T4 --
14: LD c T5 --
15: ADD T4 T5 T6
16: MUL T3 T6 T7
17: = T7 d --
18: BR 22 -- --
19: LAB 11 -- --
20: LD 9 T8 --
21: = T8 d --
22: LAB 18 -- --
23: BR 5 -- --
24: LAB 8 -- --
```

Here the Basic Blocks are (1, 4), (5, 8), (9, 11), (12, 18), (19, 21), (22, 23) and (24, -). Figure 10.1 shows the corresponding flow-graph. Note that though the branch op-code written within a BB refers to a 4-tuple entry number, in a flow-graph the branches are considered to the BB and not to a specific entry. Also the unconditional branch at the end of each BB is assumed to be present, as represented by corresponding edge.

The reasons for this convention are like this: during optimization, we may move some of the 4-tuples from one place to another within a BB or even to another BB. It will be convenient to consider the branches as referring to BB rather than a particular 4-tuple, then the movement of the 4-tuples will not affect the branch targets.

The Basic Blocks form a linked list.

10.1.2 Reference and Define Information

Suppose some value is stored in some variable or a register. The information about when it will be used in a subsequent statement is very useful in generating a good optimized code, because if the value is not going to be used later than some statement, after that statement the register holding the value can be assigned to some other variable.

For a 4-tuple `i: op X Y Z` in a BB, we say that the BB *references X* and *Y* and *defines Z*. A variable name *Z* defined in BBi is *live* if it is referenced later in some other BBj. A variable that is not live is *dead*.

If we associate with each 4-tuple in a BBi, a list of other 4-tuples in the BB where each variable is referenced, it will help us in determining the variables which are live after the end of BBi. We scan backwards from the end of the BB to prepare this list. Though to determine the live variables at the end of a BB requires *Data-flow analysis*, a simple heuristics is to assume that all user-defined variables are live and all temporaries are dead at the end of a BB. One common exception to this assumption is a loop control variable, it must be assumed live at the end of a BB which defined it.

Algorithm 10.1.2 computes the next reference information.

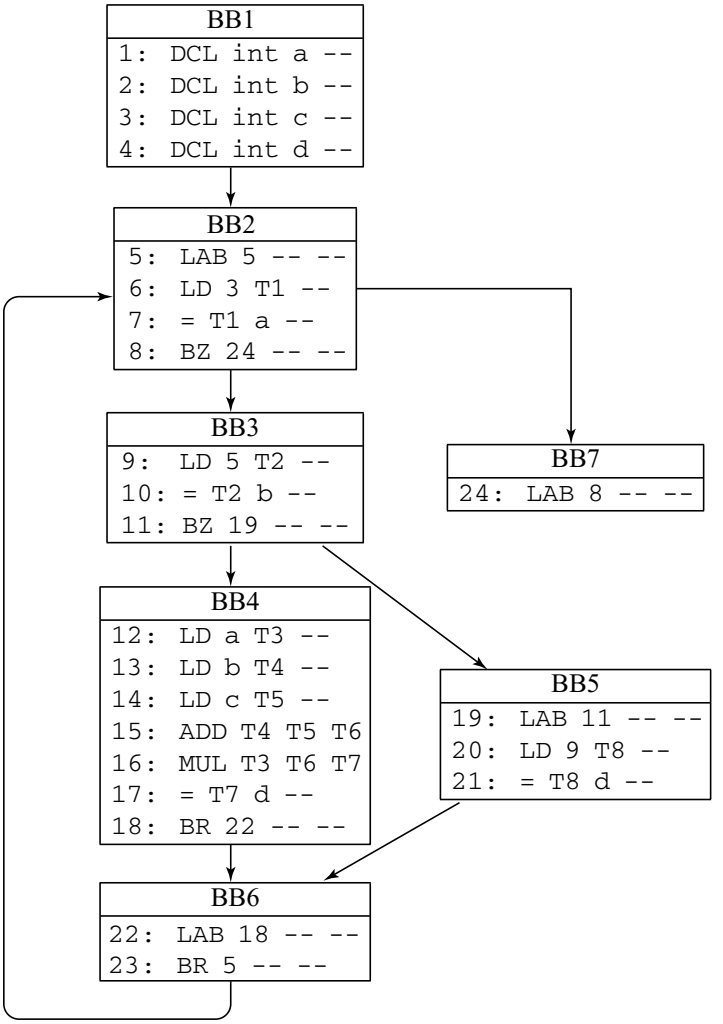


Fig. 10.1 An example Flow-graph

Algorithm 10.1.2: Next reference

/* There are N statements (i: to $N + i - 1$:) in the current BB */
mark as live in the Symbol Table each variable live at BB end;
mark next reference at $N + i$ for each such variable;
foreach 4-*tuple* [`n:op X Y Z`], $n = N + i - 1$: *to i:* **do**
  insert in Symbol Table next reference information for X, Y, Z;
  mark Z as dead in Symbol Table;
  mark X and Y live in Symbol Table, next reference n;
end

Notice that the next reference and liveness information is associated with each 4-tuple within a BB and not only the block itself.

Armed with these definitions, we can attempt various types of BB-localized optimizations. Two BB are considered equivalent if they compute the same expressions. The goal of BB-localized optimization is to get equivalent but more efficient BBs.

10.1.3 Loops in Flow-graphs

Almost all non-trivial programs in a High Level language contain looping constructs like WHILE-DO or FOR and in all such programs the major portion of execution time is spent in loop iterations. This requires that a good compiler should attempt to generate code for the loops which is as efficient as possible.

A *loop* in a flow-graph is detected as:
A set L of BB in a flow-graph is a *loop* if L contains a BB e, the *loop-entry*, such that:

1. e is not the entry to the entire flow-graph.
2. No BB in L apart from e has a predecessor outside of L. In other words, every path from the entry to the entire flow-graph to any BB within L goes through e.
3. Every BB within L has a non-empty path to e, entirely within L.

For example, the loops within the flow-graph shown in Fig. 10.1 are {BB2, BB3, BB4, BB6} and {BB2, BB3, BB5, BB6}.

10.1.4 Example Implementation – `miniC`

We implemented the Basic Block detection, their connectivity and variable liveness analysis for the `miniC` language. A C program `basicblocks.c` detects the BBs by the algorithm given in Algorithm 10.1.1, while another C program `liveness.c` computed the next reference and liveness information. It also includes the code for Basic Block detection.

Each Basic Block is represented by a Node structure:

```
typedef union
{
  char *S;
  long I;
  void *P;
} util;
typedef struct node_struct
{
  struct node_struct *link[2];
  util u, v, w, x, y, z;
} Node;
#define next(n)     ((n)->link[0])    // next BB in linked list
#define jump(n)     ((n)->link[1])    // jumped to BB
#define bb(n)       ((n)->u.S)        // BB name
#define tstart(n)   ((n)->v.P)        // ptr to first 4-tuple in BB
#define tend(n)     ((n)->w.P)        // ptr to last 4-tuple in BB
#define istart(n)   ((n)->y.I)        // index of first 4-tuple in BB
#define iend(n)     ((n)->z.I)        // index of last 4-tuple in BB
```

Each 4-tuple is represented by a Tuple structure:

```
typedef struct
{
  int i;
  char *op;
  char *opr1;
  char *opr2;
  char *opr3;
  int f1, f2, f3;  // 0: dead var, > 0: live, next ref no
  int valno;       // value number of result
} Tuple;
```

Note the three flags `f1`, `f2` and `f3`, which indicate if the corresponding variable is live and if live, the 4-tuple reference up to which its liveness is detected.

The 4-tuple IR is stored in a pointer array `Tuple *mat[1024]`, which is loaded by a function `getmat()` from the *.MAT file generated by the semantic phase.

We require a Symbol Table for our optimization operations. Had we already integrated the machine-independent optimization phase with the front-end, we could have used the main Symbol Table, but we have decided to generate the 4-tuple (and RPN) IR as text files and then try various optimization methods on them. Thus, we have to build our symbol table within the `basicblocks.c` and `liveness.c` programs. The function `mksymtab()` does that job and builds a symbol table as a hash table. It utilizes the hash table manipulation functions available in C library.

The function `mkbb()` detects and stores the Basic Blocks in the node structures, and forms a linked list of the Basic Blocks.

The function `setupbranch()` establishes the *jump* links, so that we can follow the BBs according to the execution flow.

Basic blocks and their connectivity: The following results are obtained for the example 4-tuple IR shown in Section 10.1.1. In the following, `next` is link to BB number in the linked list enumerating them and `jump` is the link to conditional or unconditional branch.

```
[1: DCL int(3) a(4) --(0)]   [4: DCL int(4) d(4) -]  next=(BB2)  jump=(BB2)
[5: LAB 5(0) --(0) --(0)]    [8: BZ 24(0) --(0) --]  next=(BB3)  jump=(BB7)
[9: LD 5(0) T2(10) --(0)]    [11: BZ 19(0) --(0) -]  next=(BB4)  jump=(BB5)
[12: LD a(18) T3(17) --(0)]  [18: BR 22(0) --(0) -]  next=(BB5)  jump=(BB6)
[19: LAB 11(0) --(0) --(0)]  [21: = T8(0) d(21) --]  next=(BB6)  jump=(BB6)
[22: LAB 18(0) --(0) --(0)]  [23: BR 5(0) --(0) --]  next=(BB7)  jump=(BB2)
[24: LAB 8(0) --(0) --(0)]   [24: LAB 8(0) --(0) -]  next=(nil)  jump=(nil)
```

The next reference and liveness analysis is done by the function `liveness()`.

Liveness analysis results from `liveness.c`:

```
[1:  DCL int(0) a(4) --(0)]
[2:  DCL int(0) b(4) --(0)]
[3:  DCL int(0) c(4) --(0)]
[4:  DCL int(0) d(4) --(0)]
[5:  LAB 5(0) --(0) --(0)]
[6:  LD 3(0) T1(7) --(0)]
[7:  = T1(0) a(8) --(0)]
[8:  BZ 24(0) --(0) --(0)]
```

```
[9:  LD 5(0) T2(10) --(0)]
[10: = T2(0) b(11) --(0)]
[11: BZ 19(0) --(0) --(0)]
[12: LD a(18) T3(17) --(0)]
[13: LD b(18) T4(17) --(0)]
[14: LD c(18) T5(17) --(0)]
[15: ADD T4(0) T5(0) T6(17)]
[16: MUL T3(0) T6(0) T7(17)]
[17: = T7(0) d(18) --(0)]
[18: BR 22(0) --(0) --(0)]
[19: LAB 11(0) --(0) --(0)]
[20: LD 9(0) T8(0) --(0)]
[21: = T8(0) d(21) --(0)]
[22: LAB 18(0) --(0) --(0)]
[23: BR 5(0) --(0) --(0)]
[24: LAB 8(0) --(0) --(0)]
```

The output shown above is in the form of the complete 4-tuple matrix. We could have also printed out the same information BB-wise. The next reference data is shown in round parentheses. Following the heuristics given above, the user-defined variables `a`, `b`, `c`, `d` are shown live throughout. The temporary variables, T1 to T8, become *dead* once their values are already absorbed in some other variable and it is not going to be used later. For example, T2 becomes dead at entry 10, as its value is assigned to *b*. Note that the liveness tagging is BB-wise.

With the information provided by the above analysis, we have a framework ready for the actual optimization steps which follow.

One simple optimization that we can do is to note that we can work with much fewer temporaries than the simple-minded sequential numbers that we used above. For example, after entry 7, T1 is dead and after entry 10, T2 is dead. These can be reused to store some other numbers. Instead of using T3, T4 and T5, we can use T1, T2 and T3. This will reduce the required number of processor registers.

10.2 Value Numbering Scheme

One particular scheme we present here is called *value numbering* scheme, which helps in redundant expression elimination and constant folding. As an additional bonus it provides information necessary for global optimization.

Consider a C source code:

```
a = 4;
k = i * j + 5;
l = 5 * a * k;
m = i;
b = m * j + i * a;
```

This may give the following 4-tuple sequence:

```
1:  LD 4 T1 --
2:  = T1 a --
3:  MUL i j T2
4:  ADD T2 5 T3
```

```
5:  = T3 k --
6:  MUL 5 a T4
7:  MUL k T4 T5
8:  = T5 l --
9:  LD i T6 --
10: = T6 m --
11: MUL m j T7
12: MUL i a T8
13: ADD T7 T8 T9
14: = T9 b --
```

The main data structure used in the value numbering method is a hash-coded table of *available expressions*. As each 4-tuple is encountered from the start of a Basic Block, this table is searched for a previous instance of the same expression. If found, all the subsequent instances can be replaced by a reference to the first one.

In order for this scheme to work, we have to determine when two operands are holding identical values. This is provided by a system of *value numbers*, where each unique value generated or used within the block is assigned an identifying number. Two entities have the same value number if and only if, based upon the information from the block alone, their values are provably identical. For example, after the second 4-tuple is scanned, the value numbers of constant 4, T1 and variable `a` are the same.

The current value number of variable or a constant is kept in the Symbol Table. The value number of result of a 4-tuple is kept in the table of available expressions and as an auxiliary field in the 4-tuple itself. The hash function for entry to the table of available expressions is based on value numbers of the operands and a code for the operator.

Constant folding is handled via a flag in the Symbol Table entry, indicating whether that entry refers to a constant and a flag in the 4-tuple to indicate if the result is a constant. We also need a table of constants, indexed by the value number and holding actual constant values.

Algorithm 10.2.1 gives an algorithm to handle value numbering scheme. Inputs to the algorithm are a Basic Block of 4-tuples and a Symbol Table. Output from the algorithm is an improved Basic Block. It uses two tables – a table of available expressions `availtab` and a table of constant values `consttab`.

The following actions take place when this algorithm is applied to the example 4-tuple IR given above:

1: to 5: Value numbers are assigned to variables `a`, `i`, `j`, `k` and to constants 4 and 5. The results of 3: and 4:, i.e. T2 and T3 are recorded as *available*. In the Symbol Table, the assigned value numbers are as follows:

Name	valno	const?
4	1	yes
a	1	yes
i	2	no
j	3	no
5	5	yes
k	6	no

The auxiliary field of 4-tuples have:

Algorithm 10.2.1: ValueNumbering(BB)

```
while there is another instruction do
    instr ← next 4-tuple;
    op ← op(instr);
    if op = '=' then
        get a1 = value number of opr1(instr);
        if a1 is a constant then
            indicate that in SymTab entry of opr2(instr);
        end
    else
        get value numbers a1 = opr1(instr), a2 = opr2(instr);
        if a1 and a2 are constants then
            compute x = op(consttab[a1], consttab[a2]);
            insert x in consttab, with a new value number;
            delete instr from the BB;
        else
            look-up 4-tuple (op, a1, a2) in availtab and set found = True if successful;
            if found then
                record the fact that any reference to this 4-tuple is to be subsumed by a reference
                to one available in availtab;
                delete instr from the BB;
            else
                enter 4-tuple (op, a1, a2) in availtab, assigning a new value number to the
                result;
            end
        end
    end
end
```

Result	valno	const?
T1	1	yes
T2	4	no
T3	6	no

The `availtab` looks like:

4-tuple	op	opr1	opr2	opr3
3	MUL	2	3	4
4	ADD	4	5	6

And the `consttab` contains (valno, value) = (1, 4) and (5, 5).

6: The algorithm looks up constant 5 and `a`, finds that both of them are constants. The result 20 is computed and a new (valno, value) pair (7, 20) is inserted in `consttab`. The 4-tuple 6 is deleted.

7: Modified to use constant 20 in place of T4.

8: to 10: The information collected is:

In the Symbol Table:

Name	valno	const?
4	1	yes
a	1	yes
i	2	no
j	3	no
5	5	yes
k	6	no
20	7	yes
l	8	no
m	2	no

The auxiliary field of 4-tuples have:

Result	valno	const?
T1	1	yes
T2	4	no
T3	6	no
T4	7	yes

The `availtab` looks like:

4-tuple	op	opr1	opr2	opr3
3	MUL	2	3	4
4	ADD	4	5	6
7	MUL	7	6	8

The `consttab` contains (valno, value) = (1, 4), (5, 5) and (7, 20).

11: The algorithm finds that value numbers for variables `m` and `j` are 2 and 3, respectively, and there is an available computation of these values as 4-tuple number 3. Thus, 4-tuple 11 is deleted and subsequent references to its result T7 are replaced by T2.

Optimized final code is shown below:

```
1:  LD 4 T1 --
2:  = T1 a --
3:  MUL i j T2
4:  ADD T2 5 T3
5:  = T3 k --
7:  MUL 20 k T5
8:  = T5 l --
9:  = i T6 --
10: = T6 m --
12: MUL i a T8
13: ADD T2 T8 T9
14: = T9 b --
```

It is interesting to note that 4-tuple 11 is found out to be same as 4-tuple 3, even though the variable names were different.

The above method can be further developed to be able to handle array references and data structures.

Further, such methods, which do local analysis limited to a Basic Block, can be modified to provide useful information for global analysis.

10.3 Peep-hole Optimization

This kind of optimization is done by one or more passes over the entire IR, in a linear fashion. Such methods are called *peep-hole* optimization because it inspects and manipulates a small portion of the IR at a time, see Fig. 10.2. A peep-hole method:

- looks for certain patterns and replaces them with simpler, or more efficient ones;
- requires only small amount of additional memory for its functioning;
- is relatively small and fast;
- works best with 3-tuple and 4-tuple IR;
- a number of individual methods are available under the umbrella of *peep-hole* optimization.

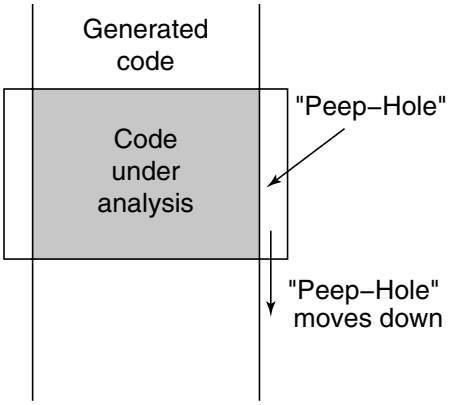


Fig. 10.2 Peep-hole Optimization

We should note here that some of the Peep-hole Optimization methods are also applicable as machine-dependent methods. Specifically the following are mostly used:

1. Removal of redundant Load/Store instructions.
2. Removal of unreachable code.
3. Removal of redundant GoTo (Branch) instructions.
4. Arithmetic manipulations like constant folding, strength reduction and algebraic simplification.
5. Processor specific code transformations, for example use of `leal` instruction in `x86` processors for doing MUL and ADD within a single instruction.

We shall now discuss these methods in some details.

10.3.1 Strength Reduction

This method replaces one operation with another requiring less time on any normal processor. It is machine independent (MI) and provides Time-space trade-off, i.e. reduction in execution time of the target code may be at the price of possible increase in memory requirements. Table 10.1 shows some of the usual strength reduction operations. The strength reduction can be applied in slightly more complex way, consider for example the following C source code:

```
i = 1;
while(i < 100){
    // some code
    a = i * 5;
    // some code
    i++;
}
```

Table 10.1 Strength reduction transformations

Operation	Replaced by	Remarks
`x * 4`	`x << 2`	*T* and *S*
`x / 8`	`x >> 3`	*T* and *S*
`x % 8`	`x & 7`	*T* and *S*
`x * 11`	`T1 = x; T2 = T1<<3; T3 = T1<<1; T1 = T1 + T2;`	Time space
	`T1 = T1 + T3; a = T1;`	Trade-off
`x && y`	If *x* is False, do not evaluate *y*	Time
`x || y`	If *x* is True, do not evaluate *y*	Time

The value of `i * 5` can be saved in a temporary `t`, whose value is incremented by 5 for each iteration of the loop, as

```
i = 1;
t = 5;
while(i < 100){
   // some code
   a = t;
   // some code
   i++; t += 5;
}
```

This replaces a multiplication by an addition.

10.3.2 Constant Folding

In this method, the compiler itself does as much of the computation as possible. If any expression or a part thereof contains only numeric constants, then the compiler can easily calculate the value at compile-time and generate the code only for the resultant value. Even the Parser, before the semantic phase, can do this computation. For example, an expression `a + 3 * 7` is replaced by `a + 21`. This is called *Constant Folding*. This action can be combined with strength reduction, for example an expression `a * 2 + 3 * 5` can be reduced to `a << 1 + 15`, where the first reduction is by strength reduction and the second by constant folding. This method saves both Time and Space.

10.3.3 Constant Propagation

Suppose we had 4-tuples like:

```
i: = 5 x --
... ... ...   // x is not assigned a new value here
j: = x y --
```

can be replaced by:

```
i: = 5 x --
... ... ...   // x is not assigned a new value here
j: = 5 y --
```

because on most processors, loading a constant in a variable is faster than loading a value from the memory. The method saves Time, but may also save Space if the constants are small.

Multiplication or Division by 1 and Addition or Subtraction of 0 can be completely eliminated.

10.3.4 Dead Variable and Dead Code Elimination

Some portion of the code may not have any chance of having any effect of the result to be computed. This portion of the code is called *Dead code*.

Dead Variable

If a variable is declared and initialized, but later in the code it does not appear in any RHS of an assignment, as a function call argument, as limit in a for or while loops, etc. till a certain statement is reached, where it is again assigned a value, then between the initialization and this statement the variable is dead.

Dead Code

Dead Code is one which cannot be reached from the beginning of the program, under any circumstances or does not do any thing useful. Several examples can be given. Consider a code fragment: `if(n = 0){do something;} next code`. Here the statements corresponding to "do something" have no chance of being executed and hence can be eliminated from the IR. (A real compiler would give a warning that "do something" is being eliminated – may be the programmer made a typing mistake, it was meant to be `if(n == 0)`.)

A combination of constant propagation and dead variable or code removal may be indicated. For example,

```
f1(){
  int i, j, a[10];
  ...  ...  ...
  i = 3;           \
  ++i;              | i is not dead
  ...  ...  ...     |
  j = a[i];        /
  ...  ...  ...
}
```

If constant propagation is used first:

```
f1(){
  int i, j, a[10];
  ...  ...  ...
  i = 3;          \
  ++i;             | i is dead as it
  ...  ...  ...      | can be replaced
  j = a[4];       /  by 4
  ...  ...  ...
}
```

and removal of the dead code:

```
f1(){
  int j, a[10];
  ...  ...  ...
  j = a[4];
  ...  ...  ...
}
```

One important aspect of dead variable or code removal is that if a variable represents result of some hardware action, e.g. a flag set or reset by a hardware device, then it can get modified due to an Interrupt Service Routine. To deal with such situations, ANSI C provides a keyword `volatile`, which declares a variable whose value may not remain constant.

10.4 Structural Optimization

In this category of optimization methods, the overall structure of the code is analyzed. Generally, such optimization results in reduction of both Time and Space. For the best optimization, a full IR in the form of AST is required.

10.4.1 Redundant Sub-expressions

Also called Elimination of common sub-expressions, this method does a pattern analysis of the IR and searches for expressions which are used repeatedly. They need to be computed only once and then the result is utilized wherever they are used later. For example, `a * b + a * b`. The 4-tuple matrix is:

```
i: LD  a  T1  --
   MUL b  T1  T2
   LD  a  T3  --
   MUL b  T3  T4
   ADD T2 T4  T5
```

and can be reduced to:

```
i: LD  a  T1  --
   MUL b  T1  T2
   ADD T2 T2  T3
```

Here, the common sub-expression was in the same main expression, but a good structural optimizer would detect and eliminate them across different expressions. For example, consider

```
c = a * b;
...  ...  ...
d = a * b + a * b
```

This can be optimized to:

```
c = a * b;
...  ...  ...
d = c << 1;
```

If the common sub-expression invokes a function call, then the speed gain is even higher; consider

```
c = sqrt(1 - sin(x) * sin(x));
```

which can be rewritten as

```
t = sin(x);
c = sqrt(1 - t * t);
```

10.4.2 Loop Unwinding

Replace a loop over a small number of iterations by equivalent linear code. For example,

```
for(i = 0; i < 5; i++) a[i] = i;
```

is replaced by

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
etc.
```

Here the compiler can pre-compute the addresses *a*[0], *a*[1], etc. and avoid generation of run-time array addressing code. Also, the loop set-up and loop-control are eliminated as well.

10.4.3 Replace Index by Pointers

In this method, we replace code segments like `a[i]` by `*(a + i)`, `b[i][j]` by `*(b + jmax * i + j)`, which is much faster, because the compiler can take advantage of constant propagation, folding, etc. The advantage is more evident if a complete array is being traversed, because the run-time indexing code can be replaced by incrementation of the pointer value. However on processors like x86, where indexing by more than one register is available, this method does not hold much significant advantage.

10.4.4 Code Motion

Moving a code portion not dependent on the iterations of a loop, called *loop-invariant code*, out of the loop can result in improved efficiency. For example,

```
for(i = 0; i < 100; i++) a[i] = numer/denom;
```

Here, the RHS of the assignment is a loop-invariant, so it can be moved out of the loop:

```
temp = numer/denom;
for(i = 0; i < 100; i++) a[i] = temp;
```

10.4.5 Variable Folding

Instructions of the form `a = b;` will become redundant if we replace all subsequent `a` by `b`, till `b` is changed or `a` is processed further.

10.4.6 In-line Function

A function call can be replaced by the body of the function, if it is short enough. The deciding factor is the cost of the code in the calling convention. It is quite a common practice in C programming, for example,

```
#define min(x,y) ((x) < (y)?(x):(y))
```

where we have defined a macro which will be reproduced everywhere `min(a,b)` is called.

10.4.7 Register Allocation

Though this method is really a structural optimization method, it is machine-dependent optimization. The idea is:

- Put temporary values which are being used frequently in processor registers.
- Save registers on a subroutine call and restore then on return from subroutine.
- *Life-time* analysis is required.

10.5 Global Data-flow Analysis

The local code optimization methods discussed till now, though quite effective, do have their limitations. The limitations arise mainly due to:

- incomplete information about the liveness of variables at the end of a Basic Block;
- impossibility of optimization across more than one Basic Block.

More advanced optimization techniques, known as *Global optimization*, mostly done via global data-flow analysis, make available more accurate information about the variables at start and end of each Basic Block. They allow for better optimization by:

- Having more accurate information, the methods discussed above, with certain assumption about liveness of variables, can be applied more effectively.
- The methods can be extended to be applied across more than one Basic Block.
- Entirely new optimization methods, applicable to loops, can be developed and used. If you remember that most of the time-wise inefficiency in a typical program arises out of loops, then the importance of such an optimization will be self-evident.

The global analysis has to answer the following problem: given a *control flow* structure, we must find out the *data flow*, i.e. which definitions of the program quantities can affect which of its uses. The questions about the data flow come under two classes:

1. Given a point in a program, what can happen before the control reaches that point, i.e. what definitions can affect computations at that point. These are *forward* flow problems.
2. Given a point in a program, what can happen after the control leaves that point, i.e. what uses can be affected by the computations at that point. These are *backward* flow problems.

Collecting information necessary to solve these problems is done in two steps – *control flow* analysis and *global data-flow* analysis. The first step is comparatively easy and we have already demonstrated it.

The global data-flow analysis is not that easy and a number of methods are proposed by various researches. We shall present here only one of them. In the following discussions, the control flow of a program is represented by a directed graph $G = (N, E, n_0)$, where N is the set of nodes (representing the Basic Blocks), E is a set of edges and n_0 is the program entry node.

10.5.1 Available Expression Analysis

We say that an expression is *defined* at a point if the value of that expression is computed there. An expression is said to be *killed* by a redefinition of one of its argument variables. An expression is *available* at a point `p` in G – the directed graph representing the program – if every path leading to `p` contains a prior definition of that expression, which is subsequently not killed.

In the discussions which follows, we shall use the following definitions:

Successor set $S(x)$ for a node x is defined as:

$$S(x) = \{y \in N \mid (x, y) \in E\}$$

Predecessor set $P(x)$ for a node x is defined as:

$$P(x) = \{y \in N \mid (y, x) \in E\}$$

Let `avail(b)` be set of expressions available on entry to a Basic Block `b`. We define a system of equations for `avail(b)`, $b \in N$, where N is the set of all the Basic Blocks in the program, in terms of sets which can be computed from local information.

Let `nkill(b)` be the set of expressions not killed in Basic Block `b` and `def(b)` be the set of expressions which are defined in `b`, without being subsequently killed in `b`, i.e. the set of expressions which are always available on exit from `b`. These definitions lead to the system of equations:

$$\text{avail}(b) = \bigcap_{x \in P(b)} \left(\text{def}(x) \cup (\text{avail}(x) \cap \text{nkill}(x))\right)$$

Solution to this system of equations will provide the desired global information.

10.5.2 Live Variable Analysis

A path in G is said to be X-clear if that path contains no assignments to the variable X. A variable X is *live* at a point `p` in G if there exists a X-clear path from `p` to the use of X.

Let `live(b)` be the set of variables which are live on entry to block `b`. Again we seek a system of equations for the live sets in terms of the local sets. Let `in(b)` be the set of variables which are live on entry to `b` because of use in `b`, and let `thru(b)` be the set of variables which are redefined in `b`. The following system of equations result in

$$\text{live}(b) = \text{in}(b) \cup \bigcup_{x \in S(b)} \left(\text{thru}(b) \cap \text{live}(x)\right)$$

Similar system of equations can be developed for most data-flow analysis problems.

There are several forms of solution methods for these equations available in literature; we discuss only one of them, known as *Iterative Live Analysis*.

10.5.3 Iterative Live Analysis

The simplest approach to data-flow analysis is to iterate through the nodes of the graph G, applying appropriate equations, till no changes take place. Algorithm 10.5.1 does live variable analysis.

Algorithm 10.5.1: Iterative Live Analysis

```
Input: in(b), thru(b), ∀b ∈ N;
Output: live(b), ∀b ∈ N;
foreach b ∈ N do live(b) ← in(b);
change ← True;
while change do
    change ← False;
    foreach b ∈ N do
        oldlive ← live(b);
        live(b) ← in(b) ∪ ⋃_{x∈S(b)} (thru(b) ∩ live(x));
        if live(b) ≠ oldlive then change ← True;
    end
end
```

This algorithm has time complexity of $O(n^2)$, where $n = |N|$.

10.6 Super-optimizers

Although code optimization certainly improves a code, it does not deliver an optimum final code in the sense of minimum possible time and space requirements.

Super-optimizers are an attempt to develop optimization strategies to reach this theoretical limit. As in the final analysis, the fastest or the smallest program can be written with reference to a particular processor only, such super-optimizers are in general processor specific. The general approach is to find a least-time or the shortest sequence of instructions for a particular processor which will be equivalent – in terms of its semantics – to the given raw target code. As will be evident from the example given below, the instruction sequence which emerges may look very different from the original target code.

10.6.1 Massalin's Super-optimizer

One of the best known super-optimizers is developed by Henry Massalin. It does an exhaustive search of a large number of instruction sequences, starting with single instruction, then sequences of two instructions, and so on till an optimum sequence equivalent to the given target code is found. Here by equivalence of code sequences means that for a given set of values in the processor registers and initial processor state, both the sequences leave the processor with the same values in its register and the same final state.

A little thought will tell you that the approach can quickly become limited by combinatorial explosion, because the search space grows exponentially with the number of candidate instructions in the trial lists. Some form of tree-pruning has to be used.

Massalin adopted an approach based on statistical sampling; by carefully selecting the initial values in the processor registers, the equivalence or otherwise can be determined quickly. He has given as an example the following code:

```
signum(int x){
  if(x > 0) return 1;
  else if(x < 0) return -1;
  else return 0;
}
```

A typical C compiler generates an assembly code of about 8 instructions, with 4 jump instructions. A good optimizing compiler would generate about 6 instructions with 2 jump instructions. We are concerned with the jump instructions, because in the modern pipelined processors, with instruction look-ahead, a jump instruction can degrade the performance considerably.

Though Massalin originally worked with an Motorola processor, here is an equivalent x86 processor instruction sequence from the super-optimizer, with `x` assumed to be in `eax`:

```
cdq
negl %eax
adcl %edx, %edx
```

The result is in `edx`. The most remarkable thing about this optimum code is that there are no jumps! It was reported by Massalin that the code generated by a super-optimizer makes clever use of status flags and tends not to use branches. This is a significant advantage on modern pipelined processors.

Though Massalin super-optimizer gives significantly better code, it has, at least at present, three major limitations – first, the approach is viable for short sequence of raw code only, say 8 to 12 instructions; second, the generated code has to be hand checked for correctness, as it is based on statistical tests; and third, the method requires that the candidate sequences of instructions be worked out afresh for each processor architecture separately.

10.6.2 GNU GCC Super-optimizer – GSO

The GNU GCC super-optimizer – GSO – is an extension of GNU Compiler Collection. There are many short sequences of code which occur quite frequently in the compiled code, for example absolute value or minimum of two values. The GNU super-optimizer works by precomputing optimum target code for such common sequences, called *Goal functions*.

The goal functions are specified in C as expressions relating arguments, like `v0`, `v1`, etc. to give a result in a variable called `r`. For example, the goal function `eq0`, which returns 1 if its single argument is 0 and returns 0 if the argument is non-zero, is defined as:

```
r = v0 == 0;
```

The following is a list of some of the goal functions currently supported in GSO:

eq and ne: Two-operand equality comparisons.

les, ges, lts, and gts: Two-operand signed inequality comparisons.

leu, geu, ltu, and gtu: Two-operand unsigned inequality comparisons.

eq0, ne0, les0, ges0, lts0, and gts0: Single-operand comparisons against zero. As discussed above, the unsigned comparisons against zero are degenerate.

neq, nne, nles, nges, nlts, ngts, nleu, ngeu, nltu, ngtu, neq0, nne0, nles0, nges0, nlts0, ngts0: Same as above except that they return negative one for true and zero for false.

naeq, nane, nales, nages, nalts, nagts, naleu, nageu, naltu, nagtu, naeq0, nane0, nales0, nages0, nalts0, and nagts0: Three-operand functions that compute the logical and of one operand with the negated result of the comparison of the other two operands. An example was shown at the start of this section.

peq, pne, ples, pges, plts, pgts, pleu, pgeu, pltu, pgtu, peq0, pne0, ples0, pges0, plts0, pgts0: These functions add one operand to the result of comparing the other two operands. E.g, plts0(a, b, c) would be written ((signed) $a <$ (signed) b) $+ c$ in C.

mins, maxs, minu, and maxu: Signed and unsigned min and max operations.

sgn: Returns positive one if the single operand is positive, zero if the operand is zero, and negative one if the operand is negative.

abs: Absolute value function.

nabs: Negated absolute value function.

10.7 Epilogue

Optimization is the field where most compiler research is done today. The tasks of the front-end – Scanner, Parser, Semantic Analysis – are well understood, and unoptimized code generation is relatively straightforward. Optimization, on the other hand, still retains a sizable measure of mysticism. High-quality optimization is more of an art than a science. Compilers for mature languages are not judged by how well they parse or analyze the code – you just expect it to do it right with a minimum of problems – but instead by the quality of the object code they produce.

In terms of algorithm complexity, many optimization problems are NP-complete and thus most optimization algorithms rely on heuristics and approximations. It may be possible to come up with a case where a particular optimization algorithm fails to produce better code or perhaps even makes it worse. However, the algorithms tend to do rather well overall.

It is worth noting that efficient code starts with intelligent decisions by the programmer. No one expects a compiler to replace BubbleSort with Quicksort. If a programmer uses an inefficient algorithm, no amount of optimization can make it efficient. Talking in terms of the complexity measure Big-O, a compiler can only make improvements to constant factors. However, all else being equal, you want an algorithm with low constant factors.

From the viewpoint of a debugger like `gdb`, the shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops, etc. Though it is possible to debug a program compiled with `-O` using `gdb`, conceptually you optimize a program which is known to be without bugs. The basic idea is: first get it working right and only then optimize.

? EXERCISES

1. The GNU `gcc` compiler provides a number of options to control the nature of optimization that is attempted on a given source code. Study the man-page of gcc to get acquainted with the optimization options available. Remember that some of the options are specific to a particular processor. Try to test all possible situations where various techniques of optimization are applicable. At the minimum, compile the following C programs with a command like `gcc <optimization options> -S mytest.c`, using O1, O2, O3, Os and "unwinding loops" options, to obtain the assembly language output.

   (a)
   ```
   int main(){
       int i, j, k;
       k = 5;
       i = (k + 5) * (k + 5);
       j = k + 5;
       printf("%d %d %d", i, j, k);
   }
   ```

   (b)
   ```
   int main(){
       int i, j, k;
       for(i = 0; i < 4; i++){
           j = 9;
           k = i + 4;
       }
   }
   ```

   (c)
   ```
   int main(){
       int i, j, k;
       for(i = 0; i < 4; i++){
           j = 9;
           k = i + 4;
       printf("%d %d %d", i, j, k);
       }
   }
   ```

2. Suresh wrote an optimizer for his compiler which did constant folding and propagation and transformed

   ```
         i = 0;
   LOOP: i = i + 1;
         ...   ...
         if(i < 10) goto LOOP;
   ```

   into:

   ```
         i = 0;
   LOOP: i = 1;
         ...   ...
         if(1 < 10) goto LOOP;
   ```

   What is wrong, if any, with this optimization? What is the basic problem?

3. Apply as many optimization methods as are applicable to the following 4-tuple IR code:

```
1:  SUB   4   2  T1
2:  DIV  T1   2  T2
3:  MUL   a  T2  T3
4:  MUL  T3  T1  T4
5:  ADD  T4   b  T5
6:  MUL  T3  T1  T6
7:  ADD  T6   b  T7
8:  MUL  T5  T7   c
```

FURTHER READING

1. Massalin, Henry. "Superoptimizer – A Look at the Smallest Program". Proc. ACM ASPLOS'87, Sigplan Notices 22, 10 (Oct 1987), 122–126.
2. Torbjorn Granlund and Richard Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler", PLDI '92 Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation ACM New York, NY, USA 1992.
3. Aho, Lam, Sethi and Ullman, "Compilers – Principles, Techniques and Tools" 2nd Edition, Pearson Education.
4. Bennett, J.P. "Introduction to Compiling Techniques", Tata McGraw Hill.

GLOSSARY

Basic blocks A portion of the code within a program with certain desirable properties that make it highly amenable to analysis. Basic blocks form the vertices or nodes in a control flow-graph.
Combinatorial Combinatorics is a branch of mathematics concerning the study of finite or countable discrete structures. Combinatorial problems arise in many areas including computer science.
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.
Dead variable is one that does not carry any information used subsequently.
Degenerate Reach lower order or trivial state.
Flow-graph is a representation, using graph notation, of all paths that might be traversed through a program during its execution.
Hash-coded Some data encoded by a hashing function.
Heuristics A common-sense rule or set of rules intended to increase the probability of solving some problem.
Loop-invariant code consists of statements or expressions which can be moved outside the body of a loop without affecting the semantics of the program.
NP-complete The most notable characteristic of NP-complete problems is that no fast solution to them is known. The time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.
Subsumed Contain or include; consider an instance of something as part of a general rule or principle.
Super-optimizer finds the optimal code sequence for a single, loop-free sequence of instructions. While usual optimizers just *improve* code a super-optimizer tries to find the *optimal sequence*.
Tree-pruning refers to removal of large chunks of a tree using certain strategies which decide which portions are not at all useful.

11

Overview of Processing of Some Languages

What you will learn in this chapter

- Some characteristics of Java
- Perl – A popular and powerful scripting language
- Logic Programming language – PROLOG
- An interesting language – FORTH

Key Words

Java, Perl, PROLOG, FORTH

In this chapter, we discuss characteristics of some of the computer programming languages, apart from C or C++. Each language is selected because of certain feature distinguishing that language, from the viewpoint of the language translation and run-time support.

We discuss the following languages:

Java – has grown from its initial offering as a safe language for delivering web-based *applets* to a general purpose language and associated platform J2EE, which promises several features critical to safe delivery of executables for heterogeneous operating platforms, over the Internet.

Perl – a *scripting* language, powerful enough to act as a *glue* language for creating applications from modules. It has a very powerful set of text processing facilities and interfaces quite easily with GUI, web servers, etc. on one hand and provides a simple but adequate object-oriented programming model on the other hand.

PROLOG – PROGramming in LOGic, is a language with a very simple syntax, which hides some powerful declarative programming capabilities. It is a language of choice for Artificial Intelligence problems, especially if they are text based. It is also used as a database language.

FORTH – a rather unusual functional language, with an extremely compact compiler–interpreter. It is especially popular in Robotics, Instrumentation and Control and similar applications not requiring heavy number crunching, but rather quickly modifiable programs.

11.1 Java

Java is a high-level, third generation programming language, like C, C++, Fortran, Smalltalk, Perl and many others. You can use Java to write computer applications that crunch numbers, process words, play games, store data or do any of the thousands of other things that computer software can do.

Compared to other programming languages, Java is most similar to C. However, although Java shares much of C's syntax, it is not C. Knowing how to program in C or, better yet, C++, will certainly help you to learn Java more quickly, but you do not need to know C to learn Java. Unlike C++ Java is not a superset of C. A Java compiler would not compile C code, and most large C programs need to be changed substantially before they can become Java programs.

What's most special about Java compared to other programming languages is that it lets you write special programs called *applets* that can be downloaded from the Internet and played safely within a web browser. Traditional computer programs have far too much access to your system to be downloaded and executed directly. Although you generally trust the maintainers of various ftp archives and bulletin boards to do basic virus checking and not to post destructive software, a lot still slips through the cracks. Even more dangerous software would be distributed if any web page you visited could run programs on your system. You have no way of checking these programs for bugs or for malicious behaviour before downloading and running them.

Java solves this problem by severely restricting what an applet can do. A Java applet cannot write to your hard disk without your permission. It cannot write to arbitrary addresses in memory and thereby introduce a virus into your computer. It should not crash your system.

11.1.1 Brief History

Java is a programming language created by James Gosling from Sun Microsystems in 1991. The first publically available version of Java (Java 1.0) was released in 1995. Over the time several versions of Java were released, which enhanced the language and its libraries. The current version of Java is Java 1.6, also known as Java 6.0.

From the Java programming language, the Java platform evolved. The Java platform allows some code written in a language other than the Java programming language and still runs on the Java Virtual Machine (JVM).

11.1.2 Overview

The Java programming language is defined by the Java compiler, the Java virtual machine and the Java class libraries. The JVM is a software implementation of a virtual computer that executes programs like a real machine.

The Java compiler translates a source Java code into an intermediate form called byte-codes. The JVM interprets these byte-codes and executes the program. It is written specifically for specific operating system and processor architecture.

The Java run-time environment (JRE) consists of the JVM and the Java class libraries.

11.1.3 Characteristics of Java

The Java language aims at providing a platform for writing a program once and then executes this program on multiple operating systems. To this end, Java has the following properties:

Platform independent: Java programs use the Java virtual machine as an abstraction and do not access the operating system directly. This makes Java programs highly portable. A Java program

which is standard complaint and follows certain rules can run unmodified on several platforms, e.g. Windows or Linux.

Object-oriented programming language: Except the primitive data types like integer or float, all elements in Java are objects.

Strongly typed programming language: Java is strongly typed, which means that the types of the variables used must be pre-defined and conversion to other objects is relatively strict, i.e. must be done in most cases by the programmer. There is almost no automatic type conversion as in C.

Interpreted and compiled language: Java source code is translated into byte-code which does not depend on the target platform. This byte-code will be interpreted by the JVM. The JVM contains a compiler, called Hotspot-compiler, which translates critical byte-code into native machine code.

Automatic memory management: Java manages the memory allocation and deallocation for creating new objects. The program does not have direct access to the memory. A garbage collector within the run-time system automatically recovers memory assigned to object to which no active pointer exists. The Java garbage collector runs in the background, keeping track of which objects the application no longer needs and reclaiming memory from them. This approach to memory handling is called implicit memory management because it does not require you to write any memory-handling code. Garbage collection is one of the essential features of Java platform.

The Java syntax is similar to C++. Java is case sensitive, e.g. the variables myValue and myvalue will be treated as different variables.

11.1.4 Development with Java

The programmer writes Java source code using a text editor which supports plain text. The editor may be a part of an IDE (Integrated Development Environment) which is normally used by a programmer for program development.

When the source code is ready, the programmer (or the IDE) calls the Java compiler (`javac`). The Java compiler creates platform-independent byte-codes. This byte-code is stored in ".class" files.

Byte-code can be executed by the JRE. The Java run-time environment is a program which executes the byte-code on the operating system. The JRE translates the byte-code into native code and executes it, e.g. the native code for Linux is different from the native code for Windows.

By default, the compiler puts each class file in the same directory as its source file. You can specify a separate destination directory with appropriate command option.

Classpath: The classpath defines where the compiler and interpreter look for .class files to load. The classpath in Java defines which Java class is available for your Java program. For example, if you want to use an external Java library you have to add the path to this library to your classpath to use it in your program.

11.1.5 The First Java Program

It has become a tradition to illustrate a computer programming language by writing a "Hello World" program and we follow that tradition here.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

11.1.6 Type Conversion

As mentioned above Java is a strongly typed language and if you need to change the type of variables Java generally requires an explicit conversion. The following examples illustrate such conversions.

Conversion to String

The following examples show conversion from other types to Strings:

```
// Convert from int to String
String s1 = String.valueOf(10);    // "10"
// Convert from double to String
String.valueOf(Math.PI);           // "3.141592653589793"
// Convert from Boolean to String
String s3 = String.valueOf(1 < 2); // "true"
// Convert from date to String
String s4 = String.valueOf(new Date()); // "Fri Apr 22 11:28:08 IST 2011"
```

The following examples show conversion from String to Number:

```
// Conversion from String to int
int i = Integer.parseInt(String);
// Conversion from float to int
float f = Float.parseFloat(String);
// Conversion from double to int
double d = Double.parseDouble(String);
```

We have already discussed the nature of JVM in Chapter 8 which may be consulted at this stage.

11.2 Perl

Perl is an acronym for Practical Extraction and Report Language. Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, etc. It borrows several ideas from natural languages:

- Learn it once, use it many times
- Learn as you go
- Many acceptable levels of competence
- Multiple ways to say the same thing
- Style not enforced
- Cooperative design

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it is easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules. Perl is highly portable, it runs on Windows, Linux, Mac OS, X, Solaris, *BSD, UN*X and others.

Here our intention is not to teach Perl language, but discuss some of its properties relevant to its implementation.

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a `main()` function or anything of that kind.

The "Hello World" program is simply:

```
#!/usr/bin/perl
print "Hello, world";
```

Here the first line, which is a special comment called "shebang" makes this little program directly executable as a command on Unix-like systems.

Perl is usually included as a standard component in present-day distributions of Unix-like operating systems. There is extensive documentation available for Perl, starting with `perltoc`, which is usually read via `perldoc`, which has ability to search out Perl documentation on an installation.

Perl looks rather odd on first sight. But once you have learned Perl, you will be able to write programs in Perl much faster than the other languages. This makes Perl especially suitable for writing programs which are used once only.

The following simple Perl program reads a text file from STDIN, consisting of people's names like "John Smith", each on a line of its own, and it prints them out with the first and second names have been swapped and separated with a comma (e.g. "Smith, John").

```
while(<>) {
  split;
  print "$_[1], $_[0]\n"; }
```

Compare it with a C program to do the same job.

Perl is typically implemented as partially compiled and part interpreted language. Thus, the execution of a Perl program tends to be slower than a corresponding C program. On the other hand, computers tend to get faster and faster, and writing something in Perl instead of C tends to save programmer time. Recent versions of Perl execute programs at speeds comparable to corresponding C programs.

We shall extend the "Hello World" Perl program a bit:

```
#!/usr/bin/perl
print "What is your name: ";
$name = <STDIN>;
chomp($name);
print "Hello, $name!\n";
```

`$name` is a scalar variable, i.e. it holds one value. The "dollar" sign in front indicates that this is a scalar variable. `<STDIN>` reads one line from the standard input. `chomp` is a function which strips last character if it is a *\n*, for example "John\n" is converted to "John".

Note the variable in the `print` statement, where Perl inserts its value. This is called *variable interpolation*.

Perl has three *types* only:

Scalar: Those variables holding a single value – be it an integer, floating point, a character, a string or a reference.

Array: Denoted by the variable name prefixed by `@` symbol, for example `@lines`, is an array or *list* of scalars. Individual elements of the array can be accessed by usual indexing (zero based), for example

`$lines[5]` accesses the 6th element of `@lines`. A number of useful functions are available for using the arrays as stacks or queues.

Hash: Denoted by the variable name prefixed by a `%` symbol, for example, `%pairs`, is an *associative array* of (Key ⇒ Value) pairs.

The variables need not be declared before their use, though it is possible to `use strict` pragma for ensuring that variables are declared before use. Similarly, `use warnings` will force variables initialization before use. The memory for newly created entities is allocated automatically and a garbage collector works in background for harvesting unused memory.

The variable lexical scope can be declared by `my` construct, which will limit a variable scope to the containing block.

Subroutines in Perl are named code blocks, with syntax `sub nameOfSub{ }`, with a special variable `@_` as input arguments array. The function `return()` allows you to return values, which could be a Scalar, an Array or a Hash.

An important concept in Perl is *context*, which controls interpretation of various entities. Basically there are two contexts – scalar and list, which makes it unique and more useful than most other script languages. For example, suppose we have used an array `@lines`, then if we write a statement `$nline = @lines`, we have a scalar context, forced by the LHS of the assignment. An array gives the number of elements in a scalar context which means `$nline` will be assigned the size of the array `@lines.` On the other hand, if we had a statement `($e1, $e2, $e3) = @lines;`, then we have a list context and an array gives the elements itself in a list context, which means that `$e1`, `$e2`, `$e3` will be assigned the values respectively of the first three elements of `@lines.`

Conversion from one scalar type (integer, string, float) to another happens automatically.

The arithmetic and relational operators are similar to C language, with additional operators for strings. There is a very rich set of string manipulation functions and regular expressions.

Perl has a rich set of control constructs, comparable and going beyond usual programming languages. For example, the usual IF construct can be written in two ways: `if(condition) { command }` and `command if condition;`, which makes a program easy to understand. The elements of an array or hash can be traversed very simply as:

```
for $info(@data){
    print $info,"\n";
}
```

which successively assigns values of elements of the array `@data` to `$info` and the print statement prints them. Similarly, keys and values in a hash can be accessed as:

```
for $key(sort keys %data){
    print "$key $data{$key}\n";
}
```

which extracts the keys of the hash, sorts them and then assigns their values successively to `$key`. The print statement prints each key and corresponding value in the hash, specified by `$data{$key}`.

There are even more compact functions available in Perl for processing elements of an array, for example,

```
@files = map {lc($_)} <*.java>;
```

will convert name of each Java source file in the current directory to lowercase. The curly braces may contain arbitrary operations to be performed on each element of the list (in example a list of Java source files.)

11.2.1 Perl Internals

`Perl` is not exactly an interpreter and it is not exactly a compiler: it is a byte-code compiler. It first compiles the input source code to an internal representation or *byte-code*, and then it executes the operations that the byte-code specifies on a virtual machine.

Comparing with Java's virtual machine, it is designed to represent an idealized version of a computer's processor. In Perl's case, however, the individual operations that can be performed are considerably higher level. For instance, a regular expression match is a single "instruction" in Perl's virtual machine. Perl uses a stack to co-ordinate and communicate results between operations.

The first stage is "parsing", the input to which is your Perl source code and the output is a tree data structure which represents what that code "means". One of the nodes in this tree is designated as the *start* node. Every node will have an operation to perform and contains a pointer to the node that the interpreter must execute next. Thus, the second phase of the operation is to execute the start node and follow the chain of pointers around the tree, executing each operation in the correct order.

Internal Data Types

As the tree of opcodes constituting a compiled Perl program is executed, Perl values are created, manipulated and destroyed. All the data types in Perl have corresponding data types in the C under Perl's hood.

Three C typedefs correspond to Perl's three basic data types and there are some additional types:

SV (scalar value)

AV (array value)

HV (hash value)

IV is a simple signed integer type guaranteed to be large enough to hold either a pointer or an integer.

I32 and I16 are types guaranteed to be large enough to hold 32- and 16-bits, respectively.

UV, U32 and U16 typedefs for storing unsigned versions of these last three typedefs.

All of these typedefs can be manipulated with the C functions described in the `perlguts` documentation. The behaviour of some of those functions is briefly discussed below.

- There are four types of values that can be copied into an SV: an integer value (IV), a double (NV), a string (PV) and another scalar (SV). There are dozens of functions for SVs to let you create, modify, grow and check for the truth or definedness of the Perl scalars they represent. Perl references are implemented as an RV, a special type of SV.
- When an AV is created, it can be created empty or populated with SVs, which makes sense since an array is a collection of scalars.
- The HV has associated C functions for storing, fetching, deleting and checking the existence of key/value pairs in the hash the HV represents.
- There is also a GV (glob value), which can hold references to any of the values associated with a variable identifier: a scalar value, an array value, a hash value, a subroutine, an I/O handle or a format.

For further details, see `perlguts` documentation.

11.3 PROLOG

PROLOG was invented in early seventies by Alain Colmerauer in France and Robert Kowalski in Britain. Prolog = Programming in Logic. Prolog is a declarative programming language unlike most common programming languages like C, C++, Java, which are imperative and procedural. Being a declarative language, the PROLOG programmer specifies a goal to be achieved and the PROLOG system works out how to achieve it. PROLOG can also be used to build a relational database. Some applications of PROLOG are:

- intelligent database retrieval,
- natural language understanding,
- expert systems,
- specification language,
- machine learning,
- robot planning,
- automated reasoning,
- problem solving.

11.3.1 A Short Introduction to PROLOG

PROLOG programs specify relationships among objects and properties of objects. When we say "Ramesh owns the book", we are declaring the ownership relationship between two objects: Ramesh and the book. We are declaring a *fact*. When we ask "Does Ramesh own the book?", we are trying to find out about a relationship. The relationships can also be given as rules such as Two persons are sisters *if* they are both female *and* they have the same parents. Thus, a rule allows us to find out about a relationship even if the relationship is not explicitly stated as a fact.

We have to be slightly more specific:

```
A and B are sisters if
  A and B are both female and
  A and B have the same father and
  A and B have the same mother and
  A is not the same as B
```

Thus programming in PROLOG involves declaring facts and defining rules by which further relationships can be found out. You can then ask questions to the PROLOG system regarding the objects. The facts are written in PROLOG as follows:

```
teaches(phd, "Compilers").
teaches(hbd, "Data Structures").
teaches(phd, "Operating Systems").
```

Note that names of properties or relationships begin with lowercase letters. The relationship name appears as the first term. The objects appear as comma-separated arguments within parentheses. A period "." must end a fact. The objects can also begin with lowercase letters or with digits and can be strings of characters enclosed in quotes. A fact is also called a *predicate*.

```
teaches(X, Y).
```

means that person X teaches a course Y. We define another set of facts:

```
studies(ramesh, "Compilers").
studies(suresh, "Compilers").
studies(mina, "Data Structures").
studies(gita, "Operating Systems").
studies(bhavesh, "Data Structures").
```

and one more:

```
semester(ramesh, 6).
semester(suresh, 6).
semester(mina, 4).
semester(gita, 8).
semester(bhavesh, 4).
```

We say that these facts together form PROLOG's *database*. After we have input this database, we can query the PROLOG system, and it will reply with an answer, for example,

```
teaches(phd, "Compilers").   <--- our query
true.                        <--- PROLOG's answer
```

To answer the query, PROLOG consults its database to see if this is a known fact. We say the query *succeeded* if the answer was "true". On the other hand, if the query were,

```
teaches(hbd, "Compilers").   <--- our query
false.                       <--- PROLOG's answer
```

the PROLOG system cannot find a matching entry in its database and replies accordingly. The query *failed*.

If our query were "what does hbd teach?", then our query would be written as:

```
teaches(hbd, Course).        <--- our query
Course = "Data Structures"   <--- PROLOG's answer
```

Note the word "Course" with the capital first letter. It indicates a *variable* in PROLOG. A variable is assigned values extracted from the database. If the query were,

```
teaches(phd, Course).        <--- our query
Course = "Compilers"         <--- PROLOG's answer
Course = "Operating Systems" <--- PROLOG's answer
```

Suppose the query was "Does phd teach ramesh?", the query in PROLOG would be

```
teaches(phd, Course), studies(ramesh, Course).
```

Now PROLOG has to find if this *conjunction* can be satisfied by the available data in the database, with the comma being interpreted as AND operation. The answer will be "true".

As a final example, we want to give a query "Whom does phd teach?", expressed in PROLOG as

```
teaches(phd, Course), studies(Student, Course). <--- our query
```

and the answers are:

```
Course = "Compilers"
Student = ramesh

Course = "Compilers"
Student = suresh

Course = "Operating Systems"
Student = gita
```

PROLOG solves this problem by proceeding left to right and then *backtracking*. Out of several "teaches" clauses, two are for phd – "Compilers" and "Operating Systems". With first answer "Compilers", PROLOG tries to satisfy the second part of the query and get two successes. As there are no more successes in the second half of the query, PROLOG *backtracks* and tries to get another answer to the first half of the query. It gets answer "Operating Systems". It again tries to satisfy the second half and gets the answer "gita". Based on this answer, we can define a new rule in our PROLOG database:

```
teacherof(Teacher, Student) :- teaches(Teacher, Course), studies(Student,
Course).
```

Here ":-" means "if" or "is implied by". The left-hand side is called the *head* and the right-hand side is called the *body* of the rule. With such a rule defined, we can give a query:

```
teacherof(T,S).
```

and we will get a list similar to the above, but for all the teachers.

11.4 FORTH

Forth is an unusual language. It provides an interactive development environment, which includes both an interpreter and a compiler. Forth programming style encourages you to break a problem down into many small fragments (factoring) and then to develop and test each fragment interactively. Forth advocates assert that breaking the edit-compile-test cycle used by conventional programming languages can lead to great productivity improvements. FORTH is an extensible language, in the sense that you can add your own functions (called *words*) to its *dictionary* and they become part of the language. In fact, major portions of FORTH are written in FORTH language.

The language used by most of the Laser printers – PostScript – is derived from FORTH.

There are many implementations of FORTH (and sister languages) available, most well-known being FIG-FORTH. An implementation for x86 machines is called `lina` and it is a Linux native, i.e. C-less, version of `ciforth` (common Intel Forth). It is mostly compliant with the ISO Forth standard.

Here we shall mainly refer to *Gforth* – a portable ANS Forth implementation from the GNU project. Some Forth implementations, usually early versions or those written to be extremely portable, compile threaded code (see Chapter 8), but many modern implementations generate optimized machine code like other common language compilers.

11.4.1 Hello World in Forth

The following FORTH program, which defines a *word* (a function), is the mandatory "HelloWorld" program:

```
: HELLO CR ." Hello, world!" ;
HELLO <cr>
```

The first line starting with a ':' defines a new word called `HELLO`, which prints a carriage-return, then a string "Hello, world!" and then a carriage-return is printed automatically by the FORTH system. The definition ends with the ';' character.

11.4.2 A Quick Introduction to FORTH

When you invoke the Forth, you will see a startup banner printed and nothing else. Forth is now running its command line interpreter, which is called the Text Interpreter (also known as the Outer Interpreter).

```
45 ok
```

On carriage-return, the text interpreter processes the input line. It breaks the line into groups of non-white-space characters, called token. For each token,

- It tries to treat it as a *command*. It does this by searching a *name dictionary*. If the token matches an entry in the name dictionary, it provides information that allows the text interpreter perform some actions. In Forth jargon, we say that the token names a word, that the dictionary search returns an execution token (`xt`) corresponding to the definition of the word and that the text interpreter executes the `xt`.
- If the text interpreter fails to find a match in the name dictionary, it tries to treat the token as a number in the current number base (initially 10). If the token legitimately represents a number, the text interpreter pushes the number onto a stack, called data stack.
- If the text interpreter is unable to do either of these things with any token, it discards the group of characters and the rest of the line, then prints an error message. If the text interpreter reaches the end of the line without error, it prints the status message "ok" followed by a carriage-return.

Now we try some user-defined words.

```
: greet ." Hello and welcome" ;
greet <cr>
Hello and welcome ok
greet greet <cr>
Hello and welcomeHello and welcome ok
```

Let us take another example:

```
: add-two 2 + . ;
4 add-two <cr>
6 ok
: demo 5 add-two;
demo <cr>
```

```
7 ok
9 greet demo add-two <cr>
Hello and welcome7 11 ok
```

The pair of words ":" and ";" is used to start and terminate a new definition, respectively. The first word after the ':' is the name for the new definition.

As you can see from the examples, a definition is built up of words that have already been defined; Forth makes no distinction between definitions that existed when you started the system up and those that you define yourself. The examples also introduce the words '.' (dot), "." (dot-quote) and "dup" (dewp). Dot takes the value from the top of the data stack and displays it. It is like ".s" except that it only displays the top item of the stack and it is destructive; after it has executed, the value is no longer on the stack. There is always one space printed after the number, and no spaces before it. Dot-quote defines a string (a sequence of characters) that will be printed when the word is executed. The string can contain any printable characters except "". A "" has a special function; it is not a Forth word but it acts as a delimiter (the way that delimiters work is described in the next section). Finally, "dup" duplicates the value at the top of the stack.

We already know that the text interpreter searches through the dictionary to locate names. We already have a definition called `add-two`. Let us try modifying it by typing in a new definition:

```
: add-two dup . ." + 2 =" 2 + . ; <cr>
ok
```

How does that work? The word ":" does two special things – first, it prevents the text interpreter from ever seeing the characters add-two. The text interpreter uses a variable called `>IN` (pronounced "to-in"), to keep track of where it is in the input line. When it encounters the word ":", it behaves in exactly the same way as it does for any other word; it looks it up in the name dictionary, finds its `xt` and executes it. When ":" executes, it looks at the input buffer, finds the word add-two and advances the value of `>IN` to point past it. It then does some other stuff associated with creating the new definition, including creating an entry for "add-two" in the name dictionary. When the execution of ":" completes, control returns to the text interpreter, which is oblivious to the fact that it has been tricked into ignoring part of the input line. Words like ":" that advance the value of `>IN` and so prevent the text interpreter from acting on the whole of the input line are called *parsing words*.

The second special thing that ":" does is to change the value of a variable called *state*, which affects the way that the text interpreter behaves. When Gforth starts up, state has the value 0, and the text interpreter is said to be interpreting. During a colon definition, started with a ":", state is set to `-1` and the text interpreter is said to be *compiling*.

In this example, the text interpreter is compiling when it processes the string `2 + . ;`. It still breaks the string down into character sequences in the same way. However, instead of pushing the number 2 onto the stack, it lays down (compiles) some magic into the definition of add-two that will make the number 2 get pushed onto the stack when add-two is executed. Similarly, the behaviours of '+' and '.' are also compiled into the definition.

One category of words does not get compiled. These so-called *immediate* words get executed regardless of whether the text interpreter is interpreting or compiling. For example, word ";" is an immediate word. Rather than being compiled into the definition, it executes. Its effect is to terminate the current definition, which includes changing the value of state back to 0.

When you execute "add-two", it has a run-time effect that is exactly the same as if you had typed

```
2 + . <cr>
```

outside of a definition.

In Forth, every word or number can be described in terms of two properties:

- Its interpretation semantics describe how it will behave when the text interpreter encounters it in interpret state. The interpretation semantics of a word are represented by an execution token.
- Its compilation semantics describe how it will behave when the text interpreter encounters it in compile state. The compilation semantics of a word are represented in an implementation-dependent way; Gforth uses a compilation token.

Numbers are always treated in a fixed way:

- When the number is interpreted, its behaviour is to push the number onto the stack.
- When the number is compiled, a piece of code is appended to the current definition that pushes the number when it runs. In other words, the compilation semantics of a number are to postpone its interpretation semantics until the run-time of the definition that it is being compiled into.

Words do not behave in such a regular way, but most have default semantics which means that they behave like the following:

- The interpretation semantics of the word are to do something useful.
- The compilation semantics of the word are to append its interpretation semantics to the current definition, so that its run-time behaviour is to do something useful.

The actual behaviour of any particular word can be controlled by using the words `immediate` and `compile-only` when the word is defined. These words set flags in the name dictionary entry of the most recently defined word, and these flags are retrieved by the text interpreter when it finds the word in the name dictionary.

A word that is marked as immediate has compilation semantics that are identical to its interpretation semantics. In other words, it behaves like the following:

- The interpretation semantics of the word are to do something useful.
- The compilation semantics of the word are to do something useful and actually the same thing, i.e. it is executed during compilation.

Marking a word as compile-only prohibits the text interpreter from performing the interpretation semantics of the word directly; an attempt to do so will generate an error. It is not necessary to use compile-only and it is not even part of ANS Forth, though it is provided by many implementations, but it is good etiquette to apply it to a word that will not behave correctly and might have unexpected side effects in interpret state. For example, it is only legal to use the conditional word IF within a definition. If you forget this and try to use it elsewhere, the fact that it is marked as compile-only allows the text interpreter to generate a helpful error message.

This example shows the difference between an immediate and a non-immediate word:

```
: show-state state @ . ;
: show-state-now show-state ; immediate
: word1 show-state ;
: word2 show-state-now ;
```

The word immediate after the definition of show-state-now makes that word an immediate word. These definitions introduce a new word: @ (pronounced “fetch”). This word fetches the value of a

variable and leaves it on the stack. Therefore, the behaviour of show-state is to print a number that represents the current value of state.

When you execute word1, it prints the number 0, indicating that the system is interpreting. When the text interpreter compiled the definition of word1, it encountered show-state whose compilation semantics are to append its interpretation semantics to the current definition. When you execute word1, it performs the interpretation semantics of show-state.

At the time that word1, and therefore show-state, are executed, the system is interpreting. When you press < *cr* > after entering the definition of word2, you should have seen the number −1 printed, followed by "ok". When the text interpreter compiled the definition of word2, it encountered show-state-now, an immediate word, whose compilation semantics are therefore to perform its interpretation semantics. It is executed straight away, even before the text interpreter has moved on to process another group of characters, the ";" in this example. The effect of executing it is to display the value of state at the time that the definition of word2 is being defined. Printing −1 demonstrates that the system is compiling at this time. If you execute word2 it does nothing at all.

Before leaving the subject of immediate words, consider the behaviour of ."" in the definition of greet above. This word is both a parsing word and an immediate word. Note that there is a space between ." and the start of the text "Hello and welcome", but there is no space between the last letter of welcome and the " character. The reason for this is that ." is a Forth word; it must have a space after it so that the text interpreter can identify it. The " is not a Forth word; it is a delimiter. The examples earlier show that when the string is displayed, there is neither a space before the H nor after the e. Since ."" is an immediate word, it executes at the time that greet is defined. When it executes, its behaviour is to search forward in the input line looking for the delimiter. When it finds the delimiter, it updates >IN to point past the delimiter. It also compiles some magic code into the definition of greet; the `xt` of a run-time routine that prints a text string. It compiles the string "Hello and welcome" into the memory so that it is available to be printed later. When the text interpreter gains control, the next word it finds in the input stream is ";" and so it terminates the definition of greet.

11.4.3 Summary

- Forth programs use factoring to break a problem down into small fragments called words or definitions.
- Forth program development is an interactive process.
- The main command loop that accepts input, and controls both interpretation and compilation, is called the text interpreter (also known as the outer interpreter).
- Forth has a very simple syntax, consisting of words and numbers separated by spaces or carriage-return characters. Any additional syntax is imposed by parsing words.
- Forth uses a stack to pass parameters between words. As a result, it uses postfix notation.
- To use a word that has previously been defined, the text interpreter searches for the word in the name dictionary.
- Words have interpretation and compilation semantics.
- The text interpreter uses the value of state to select between the use of the interpretation and the compilation semantics of a word that it encounters.
- The relationship between the interpretation and compilation semantics for a word depends upon the way in which the word was defined for example, whether it is an immediate word.

- Forth definitions can be implemented in Forth – called high-level definitions – or in some other way, usually a lower-level language and as a result often called low-level definitions, code definitions or primitives.
- Many Forth systems are implemented mainly in Forth.

If you want to learn and use FORTH in some application, you should start by getting familiar with the following words in FORTH:

Arithmetic: + − * / /MOD */ ABS INVERT

Comparison: MIN MAX =

Logic: AND OR XOR NOT

Stack manipulation: DUP DROP SWAP OVER

Loops and decisions: IF ELSE ENDIF ?DO I LOOP

Input/Output: . ." EMIT CR KEY

Defining words: : ; CREATE

Memory allocation words: ALLOT ,

Tools: SEE WORDS .S MARKER

More defining words: VARIABLE CONSTANT VALUE TO CREATE DOES>

Memory access: @ !

11.4.4 Vmgen

Vmgen, included in the Gforth package, is a tool for writing efficient interpreters. It takes a simple virtual machine description and generates efficient C code for dealing with the virtual machine code in various ways, in particular, executing it. The run-time efficiency of the resulting interpreters is usually within a factor of 10 of machine code produced by an optimizing compiler.

The interpreter design strategy supported by Vmgen is to divide the interpreter into two parts:

- The front-end takes the source code of the language to be implemented and translates it into virtual machine code. This is similar to an ordinary compiler front-end. Typically, an interpreter front-end performs no optimization, so it is relatively simple to implement and runs fast.
- The virtual machine interpreter executes the virtual machine code.

Such a division is usually used in interpreters, for modularity as well as for efficiency. The virtual machine code is typically passed between front-end and virtual machine interpreter in memory, like in a load-and-go compiler. This avoids the complexity and time cost of writing the code to a file and reading it again.

A virtual machine (VM) represents the program as a sequence of VM instructions, following each other in memory, similar to real machine code. Control flow occurs through VM branch instructions, like in a real machine.

In this setup, Vmgen can generate most of the code dealing with virtual machine instructions from a simple description of the virtual machine instructions, in particular:

VM instruction execution

VM code generation: Useful in the front-end.

VM code decompiler: Useful for debugging the front-end.

VM code tracing: Useful for debugging the front-end and the VM interpreter. You will typically provide other means for debugging the user's programs at the source level.

VM code profiling: Useful for optimizing the VM interpreter with super instructions, which are a combination of basic instructions.

To create parts of the interpretive system that do not deal with VM instructions, you have to use other tools (e.g., bison) and/or hand-code them. Vmgen supports efficient interpreters through various optimizations, in particular:

- Threaded code,
- Caching the top-of-stack in a register,
- Combining VM instructions into super instructions,
- Replicating VM (super) instructions for better BTB prediction accuracy.

As a result, Vmgen-based interpreters are only about an order of magnitude slower than native code from an optimizing C compiler on small benchmarks. On large benchmarks, which spend more time in the run-time system, the slowdown is often less. For example, the slowdown of a Vmgen-generated JVM interpreter over the best JVM JIT (Just-in-Time) compiler was measured at only a factor of 2–3 for large benchmarks. Some other JITs and all other interpreters were found slower than Vmgen-based interpreter.

VMs are usually designed as stack machines, passing data between VM instructions on a stack, and Vmgen supports such designs especially well. However, you can also use Vmgen for implementing a register VM and still benefit from most of the advantages offered by Vmgen. Vmgen is especially attractive for implementing software for embedded systems.

? EXERCISES

1. Collect details of as many programming languages as you can over a wide spectrum of applications – graphics, database access, numerical calculations including matrices and statistics, structural design, computer-aided design, etc. Try to find out their internal working details.
2. Collect details of as many legacy programming languages, which are given in Chapter 1 under language genealogy. Try to correlate the growth in the sophistication in programming languages and the hardware technology.
3. A programming language which allows a program to modify itself as the computation progresses seems to be quite attractive, especially in application fields like Artificial Intelligence. What do you think? What are pros and cons of this proposal?
4. A language named `LOGO` is quite popular in schools for teaching children basic concepts in programming like looping, subroutines, etc. Prepare a grammar specification of the language and a plan for implementing it. If time permits implement it.
5. Prepare a technical note on programming language Python, including its distinguishing features.

WEB RESOURCES

http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45202
http://en.wikipedia.org/wiki/Comparison_of_application_virtual_machines
http://www.horstmann.com/bigj/help/compiler/tutorial.html

12

Project: Compiler for a MiniC

What you will learn in this chapter

- Grammar of `miniC` – a language like simplified C
- Compiler for miniC language
- Essentials of x86 Instruction Set
- AT&T syntax for x86 Assembly language
- Some frequently used x86 assembly language idioms
- The structure of the Symbol Table in miniC compiler
- Symbol Table access functions
- Intermediate Representation and target code generation
- How to use `gdb` for checking and debugging FPU code?

Key Words

miniC language, miniC grammar, miniC compiler, target language, x86 assembly language, symbol table, intermediate representation, code generation

In this chapter, we try to demonstrate development of a compiler for a language similar to simplified C, which we call `miniC`. In the preceding chapters, we have discussed, in rather disjoint manner, a few points about the compiler for this language. Here, we try to present the whole story.

12.1 MiniC Language

The language we have selected has a syntax similar to C. It has been derived from the "higher order calculator", especially `HOC6` given by Kernighan and Pike in their pioneering book "Unix Programming Environment".

12.1.1 What is `HOC6`?

HOC, "high-order calculator", is a simple interpreted language for floating-point calculations. Its most common use is as a powerful and convenient calculator, interactively evaluating expressions such as `1 + 2 * sin(0.7)`. It also lets you assign values to variables, define your own functions, and use loops, conditionals and similar facilities you would expect in a programming language.

The HOC language was introduced and developed in the book *The Unix Programming Environment*, by Brian Kernighan and Rob Pike (Addison-Wesley, 1984). Countless people are indebted to this book, and its authors, for introducing them to Unix and its fantastic program development tools such as the shell, C and Yacc. Some of these people were strongly impressed by Kernighan and Pike's feat of building an interpreter so clearly, elegantly and easily, something which, before the advent of Yacc, was to be attempted only by experts.

In 1995, Kernighan and Pike released their version of HOC, under a BSD-like free software licence (`http://netlib.bell-labs.comb/~bwk/hoc.sh`). This meant that modified versions of HOC could legally be released, and indeed several such modified versions were released. There are several other derivations from `HOC6`, for example:

Extended high-order calculator The HOC version 7. Last updated: Nov 4 06:13:29 2004. See `http://www.math.utah.edu/pub/hoc/`
HOC ver. 9.2 by Nadav Y. Har'El, Last updated version 9.2, September 10, 2008. See `http://nadav.harel.org.il/homepage/hoc/`.

12.1.2 Objectives of `miniC`

The driving idea behind `miniC` is to provide a teaching–learning vehicle to students and teachers. The complete source code is being made available through the publishers on the website.

- To provide the teachers and students a language, its grammar and an implementation of a compiled, C-like language.
- To suggest various exercises to be done by students for modifications and extensions of the language and thereby become more familiar with compiler technology.
- To provide the complete source code so that the above objectives can be met.

While `HOC6` is designed as an interactive calculator, with its emphasis on numerical calculations and demonstration of program development facilities in Unix-like systems, our aim is to illustrate as much a compiler technology for a non-trivial programming language setting as possible within the space limitations of a text-book. Although we have adopted most of the grammar of the original Kernighan and Pike `HOC6`, there are significant differences also.

The main areas where `miniC` differs from `HOC6` are:

1. `miniC` is meant to be compiled into a target language.
2. `HOC6` handles only one type of data, viz., `double`, as the emphasis is on numerical calculation, `miniC` has three data types – *integers*, *floats* and *strings*. This was necessary to demonstrate the type-checking concepts.
3. The Symbol Table used in `HOC6` is a singly Linked List. In `miniC`, we use a Binary Search Tree to form the main Symbol Table and Hash Table C library functions in the optimization analysis function in `liveness.c`. The use of BST ADT allows a student to experiment with more sophisticated tree structures like AVL or Red–Black trees. The functions to manipulate such advanced tree structures are already available in Tree-base.
4. `HOC6` does not need declaration of a variable, but in `miniC` each variable has to be declared before it is used. This helps us in demonstrating type checking and error generation.
5. Though `HOC6` does generate a target code, the code is actually the call addresses for a library of C functions. In case of `miniC`, the target code we have selected is assembly language of `x86` (Section 12.4). It also generates two types of Intermediate representations – RPN and

4-tuples list. This will allow students to try various optimization techniques, beyond what we have demonstrated.

6. The function call conventions used in `miniC` are compatible with many popular C compilers. This makes functions written in `miniC` callable from a C program (and vice-versa, presently with some limitations). This not only helps debugging the code generated by our `miniC` compiler, but also opens up a method of further development of a compiler for `miniC`.
7. We made a conscious decision not to use the standard C library with the compiled `miniC` programs, though it is perfectly possible to do so. We have given a small macro-based library for minimally required functions, such as `read`, `print`, `square-root`, etc., and we expect the students to extend this library and use it with `miniC` programs.

12.2 Architecture of miniC Compiler

The overall working of `miniC` compiler is shown in Fig. 12.1.

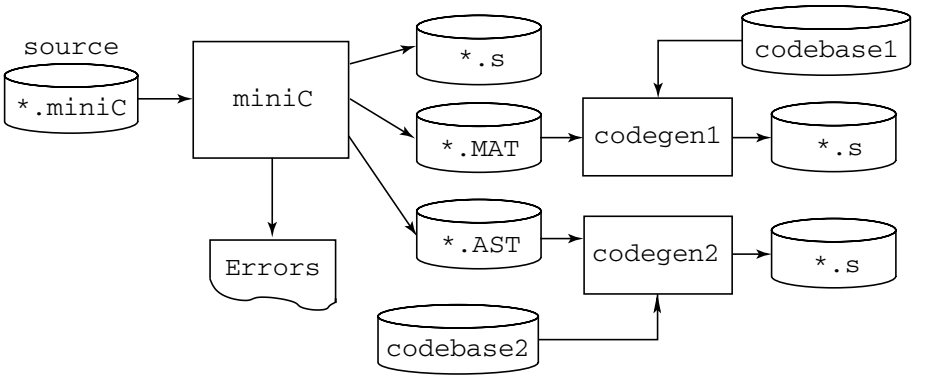


Fig. 12.1 Processing in miniC compiler. The miniC compiler provides front-end processing including semantic analysis. It outputs three types of files: (i) non-optimized assembly output, (ii) 4-tuple list output *.MAT and (iii) RPN output *.AST. The latter two allow optimization steps in the `codegen` which uses respective code pattern mapping files `codebase`. The assembly outputs are processed by system assembler `as` and linker `ld`

Further details of `miniC` front-end are shown in Fig. 12.2.

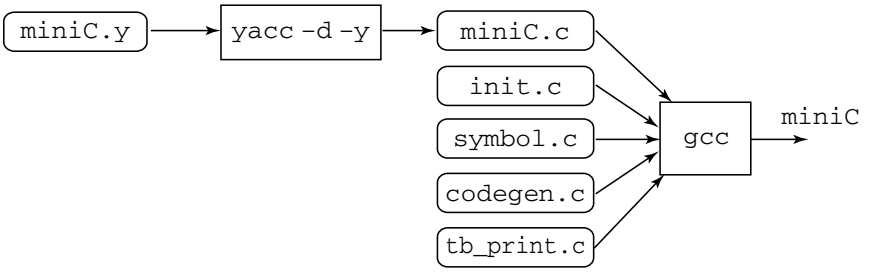


Fig. 12.2 Front-end of miniC compiler. The `codegen.c` shown here generates the non-optimized assembly code

Various components which make up `miniC` front-end are:

`miniC.c` generated from `miniC.y` – the grammar specification file – by `yacc`. Contains code for `yylex()`, several utility functions and `yacc` action terms for generating the RPN IR, 4-tuple list IR and non-optimized assembly code.

`init.c` initializes the Symbol Table by inserting pre-defined constants, keywords and built-in functions.

`symbol.c` contains functions for the Symbol Table creation, insertion and access.

`codegen.c` contains functions for generating non-optimized assembly language code for each of the parsed source language construct.

`tb_print.c` provides code to visually display the Symbol Table. This was used during the initial development of `miniC` front-end.

12.3 MiniC Grammar for `yacc`

```
%union {
Symbol *sym; /* symbol table pointer */
Symbol *inst; /* machine instruction */
int narg; /* number of arguments */
}
%token <sym>NUMBER STRING PRINT BLTIN UNDEF WHILE FOR IF ELSE
%token <sym>FUNCTION PROCEDURE RETURN FUNC PROC READ INT VAR IVAR SVAR END
%token <narg>ARG
%type  <sym>  expr
%type <inst> stmt asgn prlist stmtlist
%type <inst>cond while for if begin end
%type <sym>procname
%type <narg>arglist
%type  <sym>  dtype
%type  <sym>  dlist
%right '=' ADDEQ SUBEQ MULEQ DIVEQ MODEQ
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/' '%'
%left UNARYMINUS NOT INC DEC
%right '^'
%%
list:  stmtlist END
       | stmtlist error END
;
asgn:  VAR '=' expr
       | IVAR '=' expr
| IVAR ADDEQ expr
| IVAR SUBEQ expr
| IVAR MULEQ expr
| IVAR DIVEQ expr
| IVAR MODEQ expr
```

```
| ARG '=' expr   { }
| ARG ADDEQ expr { }
| ARG SUBEQ expr { }
| ARG MULEQ expr { }
| ARG DIVEQ expr { }
| ARG MODEQ expr { }
;
stmt:  decl
       | defn
       | expr
| RETURN
| RETURN expr
| PRINT prlist
| while'(' cond ')' stmt end
| for'(' cond ';' cond ';' cond ')' stmt end
| if'(' cond ')' stmt end
| if'(' cond ')' stmt end ELSE stmt end
| '{' stmtlist '}'
;
cond:  expr
;
while: WHILE
;
for:  FOR
;
if:  IF
;
begin:  /* nothing */
;
end:  /* nothing */
;
stmtlist: /* nothing */ }
| stmtlist '\n'
| stmtlist stmt
;
expr:  INT
       | NUMBER
| VAR
| IVAR
| ARG
| asgn
| FUNCTION begin '(' arglist ')'
| READ '(' VAR ')'
| BLTIN '(' expr ')'
| '(' expr ')'
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| expr '%' expr
```

```
| expr '^' expr
| '-' expr %prec UNARYMINUS
| expr GT expr
| expr GE expr
| expr LT expr
| expr LE expr
| expr EQ expr
| expr NE expr
| expr AND expr
| expr OR expr
| NOT expr
| INC VAR
| DEC VAR
| VAR INC
| VAR DEC
;
prlist:  expr
| STRING
| prlist ',' expr
| prlist ',' STRING
;
defn:  FUNC procname '(' ')' stmt
;
procname: VAR
| FUNCTION
;
arglist:  /* nothing */
| expr
| arglist ',' expr
;
decl:    dtype ':' dlist
       ;
dtype:   IVAR
       | VAR
       | SVAR
       ;
dlist:   VAR
       | dlist ',' VAR
       ;
%%
/* end of grammar */
```

12.4 Target Language

The target language is `x86` assembly language. We summarize here the language in the form of tables. The syntax for the assembly language used by us is known as the AT&T syntax. It is the one supported by the GNU tool chain that becomes standard with every Linux distribution. However, the official syntax for x86 assembly language (known as the Intel syntax) is different.

It is the same assembly language for the same platform, but it looks different. See Section 12.11 for some of the more important differences.

12.4.1 x86 Instructions

We do not plan to give here a description of all of hundreds of instructions in the instruction set of x86 architecture, but only those which will be used in the assembly language outputs which are given. The reader should consult appropriate documentation from the manufacturer for full details.

x86 Flags

O Overflow flag. This is set to true if the destination operand was not large enough to hold the result of the instruction.
S Sign flag. This is set to the sign of the last result.
Z Zero flag. This flag is set to true if the result of the instruction is zero.
A Auxiliary carry flag. This flag is set for carries and borrows between the third and fourth bits. It is not used by us.
P Parity flag. This flag is set to true if the low byte of the last result had an even number of 1-bit.
C Carry flag. This flag is used in arithmetic to indicate whether the result should be carried over to an additional byte. If the carry flag is set that usually means that the destination register could not hold the full result. It is up to the programmer to decide on what action to take, for example, propagate the result to another byte, signal an error or ignore it entirely.

Other flags exist, but they are much less important and not used by us.
The flags which get affected by a particular instruction are shown in the instruction set tables given below.

The source (*src*) and destination (*dst*) operands are listed giving the type of operands they take. An operand is shown as a code which tells whether the operand can be an immediate-mode value (*I*), a register (*R*) or a memory address (*M*). Note that in x86 assembly language you cannot have more than one operand, being a memory location.

Data Transfer Instructions

These instructions are mostly used for moving data from one place to another. They are given in Table 12.1.

Table 12.1 Data transfer instructions

Op	src	dst	flags	Remarks
movl	I,R,M	I,R,M	O,S,Z,A,C	This copies a word of data from one location to another.
movb	I,R,M	I,R,M	O,S,Z,A,C	Same as movl, but operates on individual bytes.
leal	M	I,R,M	O,S,Z,A,C	Takes a memory location given in the standard format, and, instead of loading the contents of the memory location, loads the computed address.
popl	R,M	—	O,S,Z,A,C	Pops the top of the stack into the given location.
pushl	I,R,M	—	O,S,Z,A,C	Pushes the given value onto the stack, pushl is a variant which pushes the current contents of the top of the stack.
xchgl	R,M	R,M	O,S,Z,A,C	Exchange the values of the given operands.

Integer Instructions

These are the basic computing instructions that operate on signed or unsigned integers, given in Table 12.2. The logic instructions are given in Table 12.3. The flow control instructions are given in Table 12.4.

Table 12.2 Integer instructions

Op	src	dst	flags	Remarks
adcl	I,R,M	R,M	O,S,Z,A,P,C	Adds the C bit and the src to the dst. If there is an overflow, sets O and C. Usually used for operations larger than a machine word. The addition on the least-significant word takes place using addl, while additions to the other words would use the adcl instruction to take the carry from the previous add into account.
addl	I,R,M	R,M	O,S,Z,A,P,C	Adds the src to the dst, storing the result in the dst. If the result is larger than the destination register, O and C are set. Operates on both signed and unsigned integers.
cdq	—	—	O,S,Z,A,P,C	Converts the %eax word into the double word consisting of %edx:%eax with sign extension. This is usually used before issuing an idivl instruction.
cmpl	I,R,M	R,M	O,S,Z,A,P,C	Compares two integers, by subtracting the src from the dst. Discards the results, but sets the flags. Usually used before a conditional jump.
decl	R,M	—	O,S,Z,A,P	Decrements the src. Use decb to decrement a byte instead of a word.
divl	R,M	—	O,S,Z,A,P	Performs unsigned division. Divides the contents of the double word contained in the combined %edx:%eax registers by the value in the src. The %eax register contains the resulting quotient, and the %edx register contains the resulting remainder. If the quotient is too large to fit in %eax, it triggers a type 0 interrupt.
idivl	R,M	—	O,S,Z,A,P	Performs signed division. Operates just like divl above.
imull	R,M,I	R	O,S,Z,A,P,C	Performs signed multiplication and stores the result in the dst. If the second operand is left out, it is assumed to be %eax, and the full result is stored in the double word %edx:%eax.
incl	R,M	—	O,S,Z,A,P	Increments the given register or memory location. Use incb to increment a byte instead of a word.
mull	R,M,I	R	O,S,Z,A,P,C	Perform unsigned multiplication. Same rules as imull.
negl	R,M	—	O,S,Z,A,P,C	Performs the two's complement of the src.
sbbl	I,R,M	R,M	O,S,Z,A,P,C	Subtract with borrowing. This is used in the same way that adc is, except for subtraction. Normally only subl is used.
subl	I,R,M	R,M	O,S,Z,A,P,C	Subtract src from dst and stores the result in the dst. This instruction can be used on both signed and unsigned numbers.

Table 12.3 Logic instructions

Op	src	dst	flags	Remarks
andl	I,R,M	R,M	O,S,Z,P,C	Logical-and of the src and dst, and stores the result in the dst. Resets O and C.
notl	R,M	—	—	Logical-not of each bit in the src. Also known as a one's complement.
orl	I,R,M	R,M	O,S,Z,A,P,C	Logical-or between src and dst and stores the result in the dst. Resets O and C.
rcll	I,%cl	R,M	O,C	Rotates the given location's bits to the left the number of times in src – either an immediate-mode value or the register %cl. C is included in the rotation, making it using 33-bits instead of 32. Also sets the overflow flag.
rcrl	I,%cl	R,M	O,C	Same as above, but rotates right.
roll	I,%cl	R,M	O,C	Rotate bits to the left, but without C.
rorl	I,%cl	R,M	O,C	Rotate bits to the right, but without C.
sall	I,%cl	R,M	C	The sign bit is shifted out to C, and a zero is placed in the least significant bit. Other bits are simply shifted to the left. This is the same as the regular shift left. The number of bits to shift is either specified in immediate mode or is contained in the %cl register.
sarl	I,%cl	R,M	C	The least significant bit is shifted out to C. The sign bit is shifted in and kept as the sign bit. Other bits are simply shifted to the right. The number of bits to shift is either specified in immediate mode or is contained in the %cl register.
shll	I,%cl	R,M	C	Shifts all bits to the left, and sign bit is not treated specially. The leftmost bit is pushed into C. The number of bits to shift is either specified in immediate mode or is contained in the %cl register.
shrl	I,%cl	R,M	C	Shifts all bits in the register to the right, and sign bit is not treated specially. The rightmost bit is pushed into C. The number of bits to shift is either specified in immediate mode or is contained in the %cl register.
testl			O,S,Z,A,P,C	Does a logical-and of src and dst and discards the results, but sets the flags.
xorl			O,S,Z,A,P,C	Exclusive-or of src and dst, and stores the result in the dst. Sets the overflow and carry flags to false.

Table 12.4 Flow control instructions

Op	src	dst	flags	Remarks
call	dst	—	O,S,Z,A,C	Pushes address of next instruction onto the stack, and jumps to the dst. Used for function calls. Alternatively, the dst can be an * followed by a register for an indirect function call.
int	I	—	O,S,Z,A,C	Causes an interrupt of the given number. This is usually used for system calls and other kernel interfaces.
jcc	dst	—	O,S,Z,A,C	Conditional jump, `cc` is the condition code. See condition codes below.
jmp	dst	—	O,S,Z,A,C	An unconditional jump. Sets %eip to the destination address. Alternatively, the dst can be an * followed by a register for an indirect jump.
ret	—	—	O,S,Z,A,C	Pops a value off the stack into %eip. Used to return from function calls.

Condition codes are:

- [*n*]**a**[*e*] – above (unsigned greater than). An `n` can be added for *not* and an `e` can be added for "or equal to"
- [*n*]**b**[*e*] – below (unsigned less than)
- [*n*]*e* – equal to
- [*n*]*z* – zero
- [*n*]**g**[*e*] – greater than (signed comparison)
- [*n*]**l**[*e*] – less than (signed comparison)
- [*n*]*c* – carry flag set
- [*n*]*o* – overflow flag set
- [*n*]*p* – parity flag set
- [*n*]*s* – sign flag set
- **ecxz** – %ecx is zero

12.4.2 Assembler Directives

These are instructions to the assembler and linker, instead of instructions to the processor. The most essential of them are given in Table 12.5.

12.4.3 Floating-point Instructions

The x86 architecture implements the IEEE-754 standard. The IEEE-754 32-bit precision format divides the 32-bit value into three different sections:

Sign size = 1-bit, bit-range: 31-31;
Biased exponent size = 8-bits, bit-range: 30-23; minimum value = −126 and maximum value = 127;
Mantissa size = 23-bits, bit-range: 22-0.

Table 12.5 Assembler directives

Op	Operands	Remarks
.ascii	Quoted string	Takes the given quoted string and converts it into byte data.
.byte	Values	Takes a comma-separated list of values and inserts them right there in the program as data.
.endr	—	Ends a repeating section defined with `.rept`.
.equ	Label, value	Sets the given label equivalent to the given value. The value can be a number, a character or a constant expression that evaluates to a number or character. From that point on, use of the label will be substituted for the given value.
.globl	Label	Sets the given label as global, meaning that it can be used from separately compiled object files.
.include	File	Includes the given file just as if it were typed in right there.
.lcomm	Symbol, size	This is used in the .bss section to specify storage that should be allocated when the program is executed. Defines the symbol with the address where the storage will be located, and makes sure that it is the given number of bytes long.
.long	Values	Takes a sequence of numbers separated by commas, and inserts those numbers as 4-byte words right where they are in the program.
.rept	Count	Repeats everything between this directive and the .endr directives the number of times specified.
.section	Section name	Switches the section that is being worked on. Common sections include `.text` (for code), `.data` (for data embedded in the program itself) and `.bss` (for uninitialized global data).
.type	Symbol, ©function	Tells the linker that the given symbol is a function.

The mantissa expresses a fraction of the form 1.x where x is expressed by the mantissa section. Let s be the sign bit, be be the biased exponent and m_i, $i \in \{1, 2, \ldots, 23\}$ be the mantissa, then the value of most of the IEEE-754 encoded floats is given by:

$$value(s, be, m) = (-1)^s \cdot 2^{be-127} \cdot \left(1 + \sum_{i=1}^{23} m_i \cdot 2^{-i}\right)$$

There are certain special cases for invalid numbers, infinity and zero, etc. There are positive and negative infinities, positive and negative NaNs (Not-a-Numbers).

The special cases are:

1. if exponent is 0 and fraction is 0, the number is ± 0 (depending on the sign bit),
2. if exponent = 255 and fraction is 0, the number is $\pm$*infinity* (again depending on the sign bit), and
3. if exponent = 255 and fraction is not 0, the number being represented is not a number (NaN).

Some sample values (in hex) are: positive zero: 00 00 00 00, negative zero: 80 00 00 00, positive 1: 3F 80 00 00, negative 1: BF 80 00 00.

There are five possible exceptions with floating-point numbers:

- Invalid operation (e.g., square root of a negative number)
- Division by zero
- Overflow (a result is too large to be represented correctly)
- Underflow (a result is very small (outside the normal range) and is inexact)
- Inexact

The FPU contains 8 80-bit stack elements called %st(0), %st(1), …, %st(7). They are implemented as a stack so that %st(0) always refers to the top element, %st(1) to the one below the top, etc. This makes it quite tricky to use.

FPU code is rather fragile and often hard to read. It is strongly recommended that you read a good instruction set reference simultaneously when you are working with FPU code. We take some easy examples to see how the FPU stack works. Here is the state of the FPU at the beginning of the program:

%st(0) undefined
%st(1) undefined
%st(2) to %st(7) undefined

All FPU instructions have the `f-` prefix. If an instruction has a `-p` suffix it means that it pops the top element off the stack at the end of its execution. We are going to execute the following instructions on the FPU now:

```
#push one on top of the stack
  fld1
#push zero on top of the stack
  fldz
#add them up and store the result in \%st(0)
  fadd %st(1), %st(0)
#add the top two elements up and store the result in \%st(1),
#pop the stack
  faddp %st(0), %st(1)
```

Let us go through the changes in the FPU stack, instruction by instruction. Changes to registers are shown in italics. We start with `fld1`, which puts constant 1.0 on the stack:

%st(0) *1.0*
%st(1) undefined
%st(2) to %st(7) undefined

Next, `fldz` adds another constant (0) to the top of the stack:

%st(0) *0.0*
%st(1) 1.0
%st(2) to %st(7) undefined

Perform an addition `%st(0) = %st(0) + %st(1)` with the specified stack elements using `fadd`:

%st(0) *1.0*
%st(1) 1.0
%st(2) to %st(7) undefined

The next instruction `faddp` does two things. The FPU performs the addition `%st(1) = %st(1) + %st(0);` using `faddp`,

%st(0) 1.0
%st(1) *2.0*
%st(2) to %st(7) undefined

and then discards the top element after the addition (the *p* suffix):

%st(0) 2.0
%st(1) to %st(7) undefined

The data types handled by the FPU are 16-, 32- and 64-bit integers, and single (32-bit), double (64-bit) and extended (80-bit) precision floating point. Each supported type has an instruction *mnemonic suffix* and a *constructor* associated with it. Instruction mnemonic suffixes specify the operand's data type. Constructors build these data types into memory.

- Floating-point constructors are '.float' or '.single', '.double' and '.tfloat' for 32-, 64- and 80-bit formats. These correspond to instruction mnemonic suffixes 's', 'l' and 't'. 't' stands for 80-bit (10-byte) real. The only FPU support for this format via the 'fldt' (load 80-bit real to stack top) and 'fstpt' (store 80-bit real and pop stack) instructions.
- Integer constructors are '.word', '.long' or '.int' and '.quad' for the 16-, 32- and 64-bit integer formats. The corresponding instruction mnemonic suffixes are 's' (single), 'l' (long) and 'q' (quad). As with the 80-bit real format, the 64-bit 'q' format is only present in the 'fildq' (load quad integer to stack top) and 'fistpq' (store quad integer and pop stack) instructions.
- Register to register operations should not use instruction mnemonic suffixes, 'fstl %st, %st(1)' will give a warning and be assembled as if you wrote 'fst %st, %st(1)', since all register to register operations use 80-bit floating-point operands. (Contrast this with 'fstl %st, mem', which converts '%st' from 80- to 64-bit floating-point format, then stores the result in the 4 byte-location 'mem'.)

Remember that the floating-point registers are 80-bit wide, while the x86 integer registers are 32-bit wide. This means that the data transfer to and from the FPU registers should always be done via the FPU load and store instructions. We give here a small example of FPU manipulations, using the standard C library `printf()` function to display the results. The name of the test program file is `test11asm.S`. This example will also illustrate how to use the C library with our own assembly language programs, mainly to do the input/output.

```
.section .data
fmt:     .asciz "number = %f\n"
newline: .asciz "\n"
number:  .float   123.456

.section .text
       .globl _start
_start:
        pushl %ebp
        movl  %esp, %ebp
        flds  number
        leal  -8(%esp), %esp
        fstpl (%esp)
        pushl $fmt
        call  printf
        addl  $8, %esp
        fld1
        fldz
        fadd  %st(1), %st(0)
        faddp %st(0), %st(1)
        leal  -8(%esp), %esp
        fstpl (%esp)
        pushl $fmt
        call  printf
        addl  $8, %esp
_exit:
   movl $1, %eax
   movl $0, %ebx
   int  $0x80
```

This program was processed by:

```
as -o test11asm.o test11asm.S
ld --dynamic-linker /lib/ld-linux.so.2 -lc -o test11asm test11asm.o
```

and when executed, it printed:

```
number = 123.456001
number = 2.000000
```

The x86 FPU has built-in instructions for computing the usual mathematical functions such as square-root, logarithm, exponential, sine, cosine, etc. We shall use them as built-in functions in the miniC language.

12.5 Symbol Table

The Symbol Table is the most important data structure initiated by the Scanner and used throughout the compilation operations. There are various ways in which a Symbol Table can be built. In case of

miniC compiler, we decided to use a Binary Search Tree (BST) as Symbol Table. In general, we do not need to delete an entry in a Symbol Table and that fact fits well with the characteristics of a BST. Each node in the Symbol Table tree has the structure:

```
typedef union{
  struct node_struct*N;
  char *S;
  long I;
  float F;
  void *P;
}util;
typedef struct node_struct{
struct node_struct *link[2]; // left = 0, right = 1
util u,v,w,x,y,z;
}Node;
typedef Node Symbol;
```

Further, the following definitions provide shortcuts to access various fields in the node.

```
#define L(n)          ((n)->link[0])
#define R(n)          ((n)->link[1])
#define name(n)       ((n)->w.S)
#define type(n)       ((n)->v.I)       // syntactic type
#define fval(n)       ((n)->u.F)
#define ival(n)       ((n)->u.I)
#define ptr(n)        ((n)->u.P)
#define defn(n)       ((n)->u.P)
#define str(n)        ((n)->u.S)
#define ttree(n)      ((n)->z.N)       // link to type tree contruct
#define subtype(n)    ((n)->y.I)       // data type
#define nxtinblk(n)   ((n)->x.N)       // next identifier in the same block
```

The Symbol Table insertion, access, etc. is done via the following functions in the source file symbol.c, essential contents of which is:

```
Symbol *symtab; /* symbol table */
Symbol* lookup(char* s){        /* find s in symbol table */
        Symbol *sp;
        sp = malloc(sizeof(Symbol));
        name(sp) = malloc(strlen(s) + 1);
        strcpy(name(sp), s);
        sp = search_node(symtab, sp);
        return sp;
}
Symbol* install(char* s, int t, double d){/* install s in symbol table */
        Symbol *sp;
        sp = malloc(sizeof(Symbol));
        name(sp) = malloc(strlen(s) + 1);
        strcpy(name(sp), s);
        type(sp) = t;
```

```
        if(t == INT || t == IVAR) ival(sp) = (int)d;
        else if(t == NUMBER || t == VAR) fval(sp) = (float)d;
        insert_node(&symtab, sp);
        return sp;
}
int comp(Node *i, Node *j){
  return strcmp(i->w.S, j->w.S);
}
void display_value(Node *n){
  if(n == NULL) printf("No value available\n");
  else printf("[%s](%d){%x}\n", n->w.S,n->v.I,n->u.P);
}
```

These functions use the basic tree-ADT functions in our *Tree-base* for BST manipulations, shown below:

```
void insert_node(Node **r, Node *n){
  if((*r) == NULL)
    {
      (*r) = n;
      L(*r) = NULL;
      R(*r) = NULL;
      return;                      /* tree was empty */
    }
  if(comp(n, (*r)) < 0) //comp(n->u, (*r)->u)
    insert_node(&L (*r), n);
  else
    insert_node(&R (*r), n);
  return;
}
void traverse_rec(Node *r);
void traverse(){
  Node *r = symtab;
  traverse_rec(r);
}
void traverse_rec(Node *r){
  if(r == NULL)
    return;
  traverse_rec(r->link[0]);
  display_value(r);
  traverse_rec(r->link[1]);
}
Node *search_node(Node *r, Node *n){
  if(r == NULL) return NULL;
  if(comp(n, r) < 0) // comp(n->u, r->u) < 0
    return search_node(L(r), n);
  else if(comp(n, r) > 0)// comp(n->u, r->u) > 0
    return search_node(R(r), n);
  else return r;
}
```

Note that though our *Tree-base* library does contain BST delete function, it is not used in the `miniC` compiler and hence not shown here.

12.6 Scanner

The Scanner `yylex()` is included in the `yacc` source file `miniC.y`. We did not use `lex` or `flex` to generate the Scanner. The Scanner is comparatively straightforward, but the handling of integer and float constants possibly requires some explanation. An integer may be a prefix in a float value, which makes the scanning somewhat tricky.

The Keywords, Built-in function names and proper variables all are detected as IDentifiers by the Scanner, using the regular expression `[a-zA-Z_][a-zA-Z_0-9]*`. Before the Scanner becomes active, the initializer function in the source file `init.c` has already inserted the Keywords and Built-in names in the Symbol Table. Thus, the Scanner is able to separate out variable proper from the Keywords and Built-in names.

Integer constants can be matched against the regular expression `[0-9]+`. A float constant can take any of the forms like 3.14 -3. .14 3e + 2 -0.314E + 1, etc.

A compact and combined regular expression for an integer and a float is:

```
(([0-9]+(.[0-9]*)?|.[0-9]+)([eE][+-]?[0-9]+)?)
```

Figure 12.3 shows a part of the DFA model of the Scanner. The Scanner code is given below:

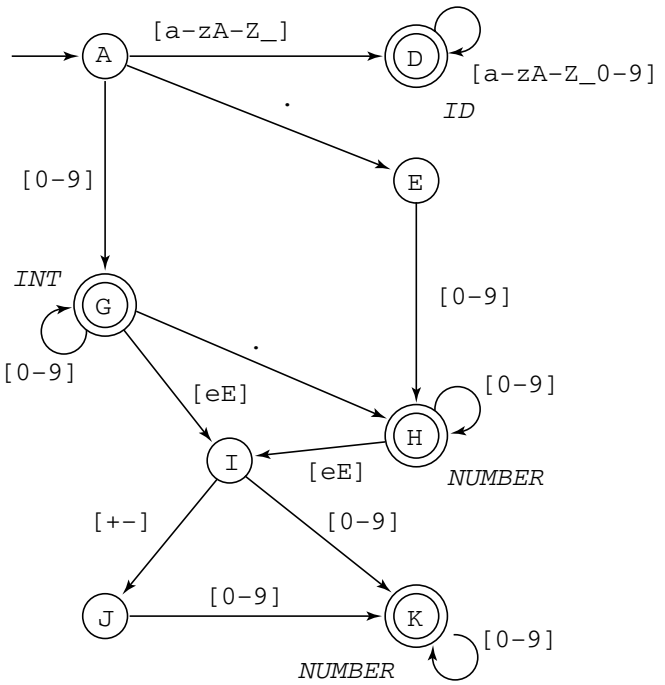


Fig. 12.3 A part of the DFA model of the Scanner. A variable type is determined by its declaration and here all the three types of variables are indicated by the token `ID`

```
yylex(void)                  /* miniC */
{
while((c = getc(fin)) == ' ' || c == '\t')
        ;
if(c == EOF)
        return 0;
if(c == '\\') {
        c = getc(fin);
        if(c == '\n') {
                lineno++;
                return yylex();
        }
}
if(c == '#') {          /* comment */
         while((c = getc(fin)) != '\n' && c != EOF)
                 ;
         if(c == '\n')
                 lineno++;
         return c;
}
if(c == '.' || isdigit(c)) {          /* number */
         double d;
         char numb[20], numb2[20];
         int myi;
         Symbol *s;
         ungetc(c, fin);
         fscanf(fin, "%[0-9.eE+-]", numb2);
         if(   strchr(numb2,'.') != NULL
            || strchr(numb2,'e') != NULL
            || strchr(numb2,'E') != NULL){
            sscanf(numb2, "%lf", &d);
            sprintf(numb,"%11.7e",(float)d);
            if((s = lookup(numb)) == 0)
               s = install(numb, NUMBER, d);
            yylval.sym = s;
            return NUMBER;
         } else {                             /* integer */
            sscanf(numb2, "%d", &myi);
            sprintf(numb,"%d" , myi);
            if((s = lookup(numb)) == 0)
                s = install(numb, INT, myi);
            yylval.sym = s;
            return INT;
         }
}
```

```
if(isalpha(c) || c == '_'){
        Symbol *s;
        char sbuf[100], *p = sbuf;
        do {
               if(p >= sbuf + sizeof(sbuf) - 1) {
                       *p = '\0';
                       execerror("name too long", sbuf);
               }
               *p++ = c;
        } while((c = getc(fin)) != EOF && (isalnum(c) || c == '_'));
        ungetc(c, fin);
        *p = '\0';
        if((s = lookup(sbuf)) == 0)
                s = install(sbuf, UNDEF, 0.0);
        yylval.sym = s;
        return type(s) == UNDEF ? VAR : type(s);
}
if(c == '$') { /* argument? */
        int n = 0;
        while(isdigit(c = getc(fin)))
                n = 10 * n + c - '0';
        ungetc(c, fin);
        if(n == 0)
                execerror("strange $...", (char *)0);
        yylval.narg = n;
        return ARG;
}
if(c == '"') { /* quoted string */
        char sbuf[100], *p;
        for(p = sbuf; (c = getc(fin)) != '"' ; p++) {
                if(c == '\n' || c == EOF)
                        execerror("missing quote", "");
                if(p >= sbuf + sizeof(sbuf) - 1) {
                        *p = '\0';
                        execerror("string too long", sbuf);
                }
                *p = backslash(c);
        }
        *p = 0;
        yylval.sym = (Symbol *)emalloc(strlen(sbuf) + 1);
        strcpy((char*)yylval.sym, sbuf);
        return STRING;
}
```

```
switch(c) {
case '+':     return follow('+', INC, follow('=', ADDEQ, '+'));
case '-':     return follow('-', DEC, follow('=', SUBEQ, '-'));
case '*':     return follow('=', MULEQ, '*');
case '/':     return follow('=', DIVEQ, '/');
case '%':     return follow('=', MODEQ, '%');
case '>':     return follow('=', GE, GT);
case '<':     return follow('=', LE, LT);
case '=':     return follow('=', EQ, '=');
case '!':     return follow('=', NE, NOT);
case '|':     return follow('|', OR, '|');
case '&':     return follow('&', AND, '&');
case '\n':    lineno++; return '\n';
default:      return c;
        }
}
backslash(int c)     /* get next char with \'s interpreted */
{
        static char transtab[] = "b\bf\fn\nr\rt\t";
        if(c != '\\')
                return c;
        c = getc(fin);
        if(islower(c) && strchr(transtab, c))
                return strchr(transtab, c)[1];
        return c;
}

follow(int expect, int ifyes, int ifno) /* look ahead for >=, etc. */
{
        int c = getc(fin);

        if(c == expect)
                return ifyes;
        ungetc(c, fin);
        return ifno;
}
```

12.7 Parser

We have discussed most of the parser algorithms in Chapters 4–9 and as such we shall not repeat those details here. The Parser itself is generated, as usual, by `yacc`.

12.8 Code Generation

We generate three kinds of outputs – raw assembly language code, reverse Polish notation (RPN) Intermediate Representation (IR) and 4-tuple IR. The code generation is achieved through `yacc` action terms.

Due to space constraints we do not propose to give the complete listings of the `yacc` action terms, but give a few indicative examples below.

12.8.1 Arithmetic Expression

We give here the `yacc` action terms for `expr '+' expr:`

```
| expr '+' expr { fprintf(fout1,"+\n"); $$ = $3;
                  fprintf(fout2,"%d: ADD T%d T%d T%d\n", ++mcount,
                           tpop(), tpop(), ++tcount); tpush(tcount);
                  efadd();}
```

The files `fout1` and `fout2` get the RPN IR and 4-tuple IR outputs, respectively. The raw assembly language code is generated by the function `efadd()` in the file `codegen.c`, which is:

```
efadd(){
  emit(" faddp\n",0);
}
```

It outputs the FPU instruction `faddp` to file `fout`.

12.8.2 Assignment

Here are the action terms for a typical assignment:

```
asgn: VAR '=' expr{fprintf(fout1,"%s\n = \n", name($1));
                   fprintf(fout2,"%d: = T%d %s --\n", ++mcount,
                                  tcount, name($1));tpush(tcount);
                   pfassgn($1);
                  }
```

The corresponding function `pfassgn()` for generating the raw assembly code is:

```
pfassgn(Node *v){
  char str[40];
  sprintf(str," fstps %s\n", name(v));
  emit(str,0);
}
```

Notice that the floating-point value is being stored single-precision and the FPU stack is popped.

12.8.3 Comparison with Logical Result

Result of an arithmetic expression comparison is a logical value, which we represent as False = 0 and True = 1. We set a byte register as per the result of the companion with one of these two values. Then we can uniformly use a JZ instruction as the building block of all our control constructs. The following sample code which does `expr > expr` comparison illustrates this idea.

```
| expr GT expr {efgt(); }
```

The corresponding raw assembly code generator function – `efgt()` – looks like

```
efgt(){
  emit(" fcomip %%st(1)\n",0);    // want  "st(1) > st(0)?"
  emit(" setc %%al\n",0);         // i.e.   TOS   < st(1)
  emit(" andl $1, %%eax\n",0);    //        1st     2nd
  emit(" fstp FPacc\n",0);
}
```

The floating point compares the instruction sets two flags C and Z. We set `al` to 1 if the C is set, i.e. TOS is less than the second operand. Remember we want to check if `al` is 0 or 1, which is achieved by the `andl` instruction. This will set up the flags again for a subsequent JZ instruction.

Other comparisons are implemented in similar ways.

12.8.4 Integer Increment and Decrement

A very common operation in C, incrementation or decrementation of an integer variable, requires some a careful implementation. When we write a variable within an expression (on the right-hand side of an assignment, for example), such as `a`, its current value is expected to be inserted in that place. When an incremented variable is used, for example, `++a`, two operations are involved – first increment `a` and then insert that incremented value at that point in the expression. If it were `a++`, first insert the value in the expression and then increment the variable.

The `yacc` action terms are:

```
| INC IVAR    {efincv($2); pivar($2); $$ = $2; } // ++a
| DEC IVAR    {efdecv($2); pivar($2); $$ = $2; } // --a
| IVAR INC    {$$ = $1; pivar($1); efvinc($1); } // a++
| IVAR DEC    {$$ = $1; pivar($1); efvdec($1); } // a--
```

The corresponding raw assembly generation functions are:

```
efincv(Node *p){
  char str[40];
  sprintf(str," incl %s\n", name(p));
  emit(str,0);
}
efdecv(Node *p){
  char str[40];
  sprintf(str," decl %s\n", name(p));
  emit(str,0);
}
efvinc(Node *p){
  char str[40];
  sprintf(str," incl %s\n", name(p));
  emit(str,0);
}
efvdec(Node *p){
  char str[40];
  sprintf(str," decl %s\n", name(p));
  emit(str,0);
}
```

12.8.5 IF-THEN-ELSE Construct

Actions for IF-THEN-ELSE construct.

```
if: IF{Symbol *i; i = tree_root("dope"); $$ = i; ++lcount; lpush(lcount);
                                                  ++lcount; lpush(lcount);}
cond: expr{int l; fprintf(fout1, "Label%d\nBZ\n", l = lpop()); lpush(l);
                 ($$) -> y.I = ++mcount;
                 fprintf(fout2, "%d: BZ %d -- --\n", mcount, 0);
                 ethen(l);}
stmt:
      | if'(' cond ')' stmt end {int l;              /* else-less if */
        fprintf(fout1, "Label%d\n:\n", l = lpop()); lpop(); /*label1 */
        fprintf(fout2, "%d: LABI %d -- --\n" , ++mcount, ($3)->y.I);
        eendif(l);
      }     /* end, if cond fails */
| if'(' cond ')' stmt end ELSE{int l,l1 ;
      l = lpop(); /* label2 */
          fprintf(fout1, "Label%d\nBR\n", (l1 = lpop())); lpush(l1);
          fprintf(fout1, "Label%d\n:\n", l); /* label2 */
          fprintf(fout2, "%d: BR %d -- --\n", ++mcount, 0);
          ($3)->x.I = mcount;
          fprintf(fout2, "%d: LABI %d -- --\n", ++mcount, ($3)->y.I);
          eelse(l,l1);
          }
          stmt end{int l;        /* if with else */
            fprintf(fout1, "Label%d\n:\n", l = lpop()); /* label2 */
            fprintf(fout2, "%d: LABI %d -- --\n", ++mcount,($3)->x.I);
            eendif(l);
          }
```

The variables `fout1` and `fout2` are handles for the Intermediate Representation output files `*.AST` and `*.MAT` containing the RPN and 4-tuple outputs, respectively. The raw assembly code is output to a file `*.S`. All these three files have names derived from the source file `myfile.miniC`.

When `if` is parsed, two label numbers and a node in Symbol Table are created. When the "cond" is parsed, a conditional branch is generated. The RPN IR and 4-tuple IR are generated within the action terms themselves, but the raw assembly code is generated by functions in the file `codegen.c`. The three functions used for IF-THEN-ELSE construct are:

```
eendif(int l){
  emit(" Label%d:\n",l);
}
ethen(int l){
  emit(" jz Label%d\n", l);
}
eelse(int l, int l1){
  emit(" jmp Label%d\n" , l1);
  emit(" Label%d:\n", l);
}
```

WHILE-DO and FOR loops are handled in a similar manner.

12.8.6 Function Definition and Call

The code for user-defined functions is to be generated in a slightly special way. The raw assembly code output for the functions is written, as an intermediate step, to a temporary file, with file handle `fout3`. All the code generated within a function definition is to be put in this file. To manage that, we have a switch variable:

```
int definition = 0;
```

For normal code, this variable has a value zero, but when the code for a function definition is being generated its value is set to 1.

The grammar and `yacc` action terms associated with function definition, function call and formal arguments of a function are:

```
defn: FUNC procname{definition = 1; type($2) = FUNCTION;
                    fprintf(fout1,"%s\nDFN\n",name($2));
                    fprintf(fout2,"%d: DFN %s -- --\n", ++mcount,
                                                        name($2));
                   ($2)->u.I = mcount;
                    edefn($2);}
       '(' ')' stmt{fprintf(fout1,"DFE\n");
                    fprintf(fout2,"%d: RET T%d -- --\n",++mcount,
                                                            tcount);
                    efstmt(); definition = 0;}
        ;
procname: VAR        { $$ = $1;}
        | FUNCTION   { $$ = $1;}
        ;
```

An identifier is initially assigned a type VAR by the Scanner, but when it appears within a valid function definition as the function name, its type is set to FUNCTION. The function prologue code is generated by `edefn()`, each statement within the definition generates its own code, but remember that now it goes to the temporary file. The function epilogue is generated by `efstmt()`.

Within the function definition, its formal arguments are written as $1, $2, etc. The Scanner detects such arguments, assigns them type ARG and returns the argument number as the value of `yylval`:

```
if(c == '$')  { /* argument? */
                int n = 0;
                while(isdigit(c = getc(fin)))
                      n = 10 * n + c - '0';
                ungetc(c, fin);
                if(n == 0)
                       execerror("strange $...", (char *)0);
                yylval.narg = n;
                return ARG;
       }
```

When a formal argument appears within a function definition, the code generator function `parg()` pushes an appropriate value on the computation stack. If a value is assigned to a formal argument, it

is handled by function `pargassgn()`. Both these functions use stack addressing with respect to the Base Pointer `ebp`.

```
parg(int i){
  emit(" flds %d(%%ebp)\n", (i + 1) * 4);
}
pargassgn(int i){
  char str[40];
  emit(" fstps %d(%%ebp)\n", (i + 1) * 4);
}
```

A function call is handled by the following action terms:

```
  | FUNCTION begin '(' arglist ')'
   { fprintf(fout1,"%s\n", name($1));
     fprintf(fout2,"%d: CALL %s T%d %d\n", ++mcount, name($1),++tcount, 4 * $4);
     efcall($1,$4);
}

arglist: /* nothing */    {$$ = 0; }
       | expr             {fprintf(fout2,"%d: ARG T%d -- --\n",++mcount,
                           tpop()); $$ = 1; eargpush($1); }
       | arglist ',' expr {fprintf(fout2,"%d: ARG T%d -- --\n",++mcount,
                           tpop()); $$ = $1 + 1; eargpush($3);}
       ;
```

The function call arguments are pushed on the computation stack by code generator function `eargpush()` and `efcall()` generates the function invocation code.

```
efcall(Node *p, int q){
  char str[40];
  sprintf(str," call %s\n", name(p));
  emit(str, 0);
  emit(" addl $%d, %%esp\n", q * 4);
}
eargpush(Node *p){
  char str[40];
  if(type(p) == INT || type(p) == IVAR){
    sprintf(str,"    pushl %s\n", name(p));
  }
  if(type(p) == NUMBER || type(p) == VAR){
    sprintf(str,"    pushl %s\n", name(p));
  }
  if(type(p) == STRING || type(p) == SVAR){
    sprintf(str,"    pushl %s\n", name(p));
  }
  emit(str,0);
}
```

Finally, the `emit()` function switches the output depending upon the value of `definition`.

```
emit(char *s, int i){
  if(definition == 0)
    fprintf(fout, s, i);
  else if(definition == 1)
    fprintf(fout3, s, i);
}
```

As a small example, consider the following `miniC` program:

```
float: j
func my(){ $1 = $1 + 1.0
  return $1
}
j = 5.0
printr(my(j))
prints("\\n")
end
```

The code for the user-defined function `my()` is:

```
.text
  .globl my
  .type my, @function
my:
  pushl %ebp
  movl %esp, %ebp
  flds 8(%ebp)
  flds FC1
  faddp
  fstps 8(%ebp)
  flds 8(%ebp)
  movl %ebp, %esp
  pop %ebp
  ret
```

The “main” program is:

```
.data
Iacc: .int 0
.text
.globl  _start
_start: nop
    flds   FC2
    fstps  j
    flds   j
    pushl  j
    call   my
    addl   $4, %esp
    fstps  FPacc
    pushl  FPacc
    call   printr
```

```
    addl   $4, %esp
    movl   $SC3, %eax
    pushl  %eax
    call   prints
    addl   $4, %esp
    movl   %eax, %ebx
    movl   $1, %eax
    int    $0x80
.section .data
DEG: .float 57.295780
E:   .float 2.718282
FC1: .float 1.000000
FC2:   .float 5.000000
FPacc: .float 0.000000
GAMMA: .float 0.577216
PHI:   .float 1.618034
PI:    .float 3.141593
PREC:  .float 7.000000
SC3:   .ascii "\n\0"
float: .float 0.000000
int:   .int 0
j:     .float 0.000000
       .lcomm string, 4
```

When this program was assembled, linked and executed, it gave:

```
$test14
  6.00000E + 00
```

as it should.

12.8.7 Assembly Language Macros

Several assembly language macros are defined, mainly for interfacing with the Linux system calls. These macros are in a file named `marocs.S`. For example, the following macros are used to do read and write operations to the system console:

```
.macro read buff, buff_size
  movl $3, %eax
  movl $0, %ebx
  movl \buff, %ecx
  movl \buff_size, %edx
  int    $0x80
.endm
.macro write str, str_size
  movl $4, %eax
  movl $1, %ebx
  movl \str, %ecx
  movl \str_size, %edx
  int    $0x80
.endm
```

12.8.8 Built-in Functions Library

It was decided that we shall not use the standard C library for linking with the final compiled `miniC` program, though it is possible to do so, if we take care to use proper command-line arguments while linking the object files. This was demonstrated in Section 12.4.3.

Several functions, called built-in functions, which are callable from a `miniC` program, are defined. As the calling conventions of `miniC` is compatible with the standard C calling conventions, these functions can be tested independently of a compiled `miniC` program.

The following functions are available at present in the "built-in" library:

printi Print an integer.
getint Read an integer.
mystrlen Find length of a string.
prints Print a string.
gets Read a string.
printf Print a floating-point number.
getf Read a floating-point number.
myfabs Floating-point absolute value.
myfcos Cosine of an angle (radians).
myfsin Sine of an angle (radians).
myfsqrt Square-root.
extract Exponent of the floating-point argument.
atofproc Convert a string to a floating-point number.
ftoaproc Convert a floating-point number to a string.

We give here a small example of the function `strlen` which finds the length of the given null-terminated string.

```
#  find the length of a string
#  uses %ecx, %edi
#  %edi contains address of the string and %eax the result at end
.globl strlen
       .type strlen @function
strlen:
       pushl %ebp
       movl %esp, %ebp
       movl (maxstr), %ecx
       xorl %eax, %eax
       movl 8(%ebp), %edi
       repnz scasb (%edi)
       movl (maxstr), %eax
       subl %ecx, %eax
       decl %eax
       movl %ebp, %esp
       popl %ebp
       ret
```

Here, `maxstr` is the address of a global constant specifying the maximum string length that is handled, which is tentatively and arbitrarily set as 255.

12.8.9 A Few Example `miniC` Programs

Here, we present some example programs written in `miniC` to illustrate the syntax.

Example 1

A while loop plus floating point and string output.

```
float: a
a = 0.0
while(a < 10.0){
   printr(a)
   prints("\\n")
   a = a + 1.0 } end
end
```

When compiled, assembled, linked and executed it gave the following output:

```
0.00000E + 00
1.00000E + 00
2.00000E + 00
3.00000E + 00
4.00000E + 00
5.00000E + 00
6.00000E + 00
7.00000E + 00
8.00000E + 00
9.00000E + 00
```

Example 2

Boolean conditions and unary-minus test:

```
float: a
float: b
b = 1.2
a = 2.3
if(a > b) prints("a greater than b\\n")
if(a >= b) prints("a greater/equal b\\n")
if(a == a) prints("a equal to a\\n")
if(a != b) prints("a not equal to b\\n")
if(b <= a) prints("b less/equal a\\n")
if(b < a) prints("b less than a\\n")
b = -a
printr(b)
prints("\\n")
printr(-99.99)
prints("------ That's all.\\n")
end
```

The output is:

```
a greater than b
a greater/equal b
```

```
a equal to a
a not equal to b
b less/equal a
b less than a
-2.30000E + 00
-9.99900E + 01----  That's all.
```

Example 3

Test pre- and post-increment and decrement:

```
int : a
a = 7
printi(++a)
prints("\\n")
printi(a++)
prints("\\n")
printi(a)
printi(--a)
prints("\\n")
printi(a--)
prints("\\n")
printi(a)
prints(" that's all.\\n")
end
```

The result output is:

```
8
8
9          8
8
7 that's all.
```

12.8.10 Assembly Language Idioms

We present here some assembly language idioms which may help the reader in understanding and extending the `miniC` language.

IF-THEN-ELSE Construct

In C, an if-statement consists of three parts – the condition, the true branch and the false branch. However, since assembly language is not a block-structured language, you have to work a little to implement the block-like nature of C. For example, look at the following C code:

```
if(a == b){
  /* True Branch Code Here */
} else {
  /* False Branch Code Here */
}
  /* At This Point, Reconverge */
```

In assembly language, this can be rendered as:

```
# Move a and b into registers for comparison
  movl a, %eax
  movl b, %ebx
# Compare
  cmpl %eax, %ebx
# If True, go to true branch
  je true_branch
false_branch:
# This label is unnecessary,
# only here for documentation
# False Branch Code Here
# Jump to recovergence point
  jmp reconverge
true_branch:
# True Branch Code Here
reconverge:
# Both branches recoverge to this point
```

Since assembly language is linear, the blocks have to jump around each other. A case statement is written just like a sequence of if-statements.

Function Call

A function call in assembly language simply requires pushing the arguments to the function onto the stack in reverse order and issuing a `call` instruction. After the call is executed, the arguments are then popped back off of the stack or the stack pointer is adjusted. For example, consider the C code:

```
printf("The number is %d", 88);
```

In assembly language, this can be implemented as:

```
.section .data
text_string:
  .ascii "The number is %d\0"
.section .text
  pushl $88
  pushl $text_string
  call printf
  popl %eax
  popl %eax
#%eax is just a dummy variable.
```

Variables and Assignments

Global and static variables are declared using `.data` or `.bss` entries. Local variables are declared by reserving space on the stack at the beginning of the function. This space is given back at the end of the function. Interestingly, global variables are accessed differently than local variables in assembly language. Global variables are accessed using direct addressing, whereas local variables are accessed using base pointer `(%ebp)` addressing. For example, consider the following C code:

```
int global_var;
int foo(){
```

```
  int local_var;
  local_var = 1;
  global_var = 2;
  return 0;
}
```

This can be implemented in assembly language as:

```
.section .data
  .lcomm global_var, 4
  .type foo, @function
foo:
  pushl %ebp                  # Save old base pointer
  movl %esp, $ebp             # make stack pointer base pointer
  subl $4, %esp               # Make room for local_var
  .equ local_var, -4          # we can now use local_var to
                              # find the local variable
  movl $1, local_var(%ebp)
  movl $2, global_var
  movl %ebp, %esp             # Clean up function and return
  popl %ebp
  ret
```

What may not be obvious is that accessing the global variable takes fewer machine cycles than accessing the local variable. However, that may not matter because the stack is more likely to be in physical memory (instead of swap-disk) compared with the global variable, which might have been swapped out.

Loops

Loops work somewhat like IF-statements in assembly language – the blocks are formed by jumping around. In C, a WHILE loop consists of a loop body, and a test to determine whether it is time to exit the loop. A FOR loop is similar, with optional initialization and counter-increment sections. In C, a while loop looks like this:

```
while(a < b){
  /* Do stuff here */
}
/* Finished Looping */
```

This can be implemented in assembly language as:

```
loop_begin:
  movl a, %eax
  movl b, %ebx
  cmpl %eax, %ebx
  jge     loop_end
loop_body:
# Do stuff here
  jmp loop_begin
loop_end:
# Finished looping
```

The x86 assembly language has some direct support for looping as well. The `%ecx` register can be used as a counter that ends with zero. The loop instruction will decrement `%ecx` and jump to a specified address unless `%ecx` is zero. For example, if you wanted to execute a statement 100 times, you would do this in C:

```
for(i = 0; i < 100; i++){
  /* Do process here */
}
```

In assembly language, it can be written as:

```
loop_initialize:
  movl $100, %ecx
loop_begin:
  #
  #Do Process Here
  #
  #Decrement %ecx and loops if not zero
  loop loop_begin
rest_of_program:
  #Continues on to here
```

One thing to notice is that the loop instruction requires you to be counting backwards to zero. If you need to count forwards or use another ending number, you should use the loop form which does not include the loop instruction. For really tight loops of character string operations, there is also the `rep` instruction, which we have already used in `strlen()` implementation.

Structs

Structs are simply descriptions of memory blocks. For example, in C you can say:

```
struct person{
  char firstname[40];
  char lastname[40];
  int age;
}
```

This does not do anything by itself, except giving you ways of elegantly using 84-bytes of data. You can do basically the same thing using `.equ` directives in assembly language, like this:

```
.equ PERSON_SIZE, 84
.equ PERSON_FIRSTNAME_OFFSET, 0
.equ PERSON_LASTNAME_OFFSET, 40
.equ PERSON_AGE_OFFSET, 80
```

When you declare a variable of this type, all you have to do is reserve 84-bytes of space. So, if you have this in C:

```
void foo(){
  struct person p;
  /* Do stuff here */
}
```

In assembly language, you would have:

```
foo:
# Standard header beginning
  pushl %ebp
  movl %esp, %ebp
# Reserve our local variable
  subl $PERSON_SIZE, %esp
# This is the variable's offset from %ebp
  .equ P_VAR, 0 - PERSON_SIZE
# Do Stuff Here
# Standard function ending
  movl %ebp, %esp
  popl %ebp
  ret
```

To access structure members, you just have to use base pointer addressing with the offsets defined above. For example, in C you could set the person's age like this:

```
p.age = 30;
```

In assembly language, it looks like this:

```
movl $30, P_VAR + PERSON_AGE_OFFSET(%ebp)
```

Pointers

First, we take a look at global variables. For example,

```
int global_data = 30;
int *a = &global_data;
```

In assembly language, this can be written as:

```
  .section .data
global_data:
  .long 30
   movl $global_data, %eax
```

With assembly language you are almost always accessing memory via pointers. Direct addressing does that. To get the pointer itself, you just have to go with immediate-mode addressing.

Local variables are slightly more difficult. Here is how you take the address of a local variable in C:

```
void foo(){
  int a;
  int *b;
  a = 30;
  b = &a;
  *b = 44;
}
```

The same code in assembly language looks like:

```
foo:
# Standard opening
  pushl %ebp
  movl %esp, %ebp
# Reserve two words of memory
  subl $8, $esp
  .equ A_VAR, -4
  .equ B_VAR, -8
# a = 30
  movl $30, A_VAR(%ebp)
# b = &a
  leal A_VAR(%ebp), %eax
  movl %eax, B_VAR(%ebp)
# *b = 44
  movl B_VAR(%ebp), %eax
  movl $44, (%eax)
# Standard closing
  movl %ebp, %esp
  popl %ebp
  ret
```

As you can see, to take the address of a local variable, the address has to be computed in the same way the computer computes the addresses in base pointer addressing. There is an easier way – the processor provides the instruction `leal`, which stands for "load effective address". This lets the computer compute the address, and then load it wherever you want.

Getting GCC to Help

One of the nice things about GCC is its ability to output assembly language code. To convert a C language file to assembly, give the command:

```
gcc -S file.c
```

The output will be in `file.s`. In it most of the variable names have been removed and replaced either with locations with respect to the base pointer `%ebp` or references to automatically generated labels. You should turn off optimizations with `-O0` so that the assembly language output will follow your source code more closely. Something else you might notice is that GCC reserves more stack space for local variables than we do, and then use an AND instruction to adjust the address. This is for a double-word aligning of variables, which is expected to increase memory and cache efficiency.

12.8.11 Linux System Calls

Some of the more important system calls are given in Table 12.6. Remember that `%eax` holds the system call numbers, and that the return values and error codes are also stored in `%eax`.

Table 12.6 Important linux system calls

eax	Name	ebx	ecx	edx	Notes
1	exit	return value(int)	—	—	Exits the program
3	read	file descriptor	buffer start	buffer size	Reads into the given buffer
4	write	file descriptor	buffer start	buffer size	Writes the buffer to the file descriptor
5	open	null-terminated file name	option list	permission mode	Opens the given file. Returns the file descriptor or an error number
6	close	file descriptor	—	—	Closes the give file descriptor
12	chdir	null-terminated directory name	—	—	Changes the current directory of your program
19	lseek	file descriptor	offset	mode	Repositions where you are in the given file. The mode should be 0 for absolute positioning, and 1 for relative positioning
20	getpid		—	—	Returns the process ID of the current process
39	mkdir	null-terminated directory name	permission mode	—	Creates the given directory. Assumes all directories leading up to it already exist
40	rmdir	null-terminated directory name	—	—	Removes the given directory
41	dup	file descriptor	—	—	Returns a new file descriptor that works just like the existing file descriptor
42	pipe	pipe array	—	—	Creates two file descriptors, where writing on one produces data to read on the other and vice-versa. %ebx is a pointer to two words of storage to hold the file descriptors
45	brk	new	—	—	Sets the system break (i.e., .system the end of the data section). If the system break is 0, it simply returns the current system break
54	ioctl	file descriptor	request arguments	—	This is used to set parameters on device files. Its actual usage varies based on the type of file or device the descriptor references

12.9 Testing

Testing of the complete compiler was a long process. The following general steps were taken in the order shown:

The Scanner: `yylex()` within the `yacc` input file `miniC.y` was separately tested in three steps:

1. First, without any Symbol Table activity, just to check if it did scan and detect required Lexical atoms.
2. The Symbol Table routines were tested separately for creation insertion and access to the Symbol Table. This was simplified to some extent because we used well-tested Tree-base Tree-ADT functions.
3. The Symbol Table access statements in the Scanner were activated and Scanner plus Symbol Table were tested together.

Parser: The Parser generated by `yacc` from the attributed grammar was tested in the following steps:

1. The grammar was tested by inserting "stubs" as action terms in the basic grammar.
   (a) The basic grammar for `expr` and `stmt` was tested first. All conflict resolutions done by `yacc` were checked and adjusted where necessary in a couple of cases.
   (b) The aggregating constructs like `stmtlist`, `arglist`, etc. were tested
   (c) The control constructs like IF-THEN-ELSE, WHILE-DO, etc. were tested.
   (d) The remaining grammar was tested.
2. The stubs were replaced by actual action statements for the raw assembly output. No attempt at any kind of optimization was done, but correct instruction sequences for control constructs and function calling conventions were achieved.
3. The generated assembly code was tested by assembling and linking it via `as` and `ld`. This exercise would provide inputs for later implementations of code generators.
4. As the FPU is used extensively, `gdb` was used to check and debug the floating-point instruction sequences. See Section 12.10 for further details.

Semantic analysis: The stubs were replaced by actual action statements and tested again in the same order as above, but this time, the emphasis was on achieving correct RPN, 4-tuple and raw assembly outputs.

Code generation: The support libraries `macros.S` and `builtinsa.S` were tested. The code generators for RPN (i.e., *.AST to *.S) and 4-tuple (i.e., *.MAT to *.S) were tested.

Type checking and Optimization: Only a few of the optimization possibilities are actually implemented and they were checked the last.

12.10 Use of `gdb` to Debug the FPU Operations

The GNU debugger `gdb` is a very useful utility on Unix-like systems for debugging C, C++, assembly language and some other programs (with limitations). Our interest in it arises because we are using floating-point instructions to implement many of the constructs in `miniC` and `gdb` has good facility for monitoring the activity of the x86 FPU.

We do not plan to discuss here the full working of GNU `gdb`, but we demonstrate only its usefulness for checking and debugging the floating-point code generated by our compiler. We used `gdb` extensively while developing the action terms (attribute rules) in the `miniC yacc` grammar,

the code generator and the assembly language built-in library. We shall demonstrate the use of `gdb` by a small example.

12.10.1 An Example `miniC` Program

A small example program is as follows:

```
float: a,b
a = 1.1111
b = -2.0
printr(a * b)
end
```

This program, though simple, checks several aspects of `miniC`:

- Type declaration and type setting of the variables.
- Ability to handle variable list in declarations.
- Memory locations assignment to variables.
- Detection and conversion of floating-point numbers (in contrast to integers).
- Checking unary-minus operation.
- Working of assignment statement.
- Checking arithmetic expression evaluation.
- Proper argument passing to a function.
- Ability of the `printr` to print a floating-point number.
- Program termination detection.

The program file – `test8a.miniC` – was compiled by a command:

```
miniC test8a.miniC
```

This gave three output files: `test8a.S`, `test8a.AST` and `test8a.MAT`, out of which we are concerned here with the assembly language output file `test8a.S` only. It looks like:

```
.text
  .globl _start
_start:
  nop
  flds  FC1
  fstps a
  flds  FC2
  fstps b
  flds  a
  flds  b
  fmulp
  fstps FPacc
  pushl FPacc
  call  printr
  addl  $4, %esp
  movl  %eax, %ebx
  movl  $1, %eax
```

```
  int   $0x80
.section .data
  DEG:    .float 57.295780
  E:      .float 2.718282
  FC1:    .float 1.111100
  FC2:    .float -2.000000
  FPacc:  .float 0.000000
  GAMMA:  .float 0.577216
  PHI:    .float 1.618034
  PI:     .float 3.141593
  PREC:   .float 7.000000
  a:      .float 0.000000
  b:      .float 0.000000
  float:  .float 0.000000
  int:    .int 0
```

Notice that the two constants 1.1111 and −2.000 have been stored as if they are variables with generated names `FC1` and `FC2`. Others are mostly `miniC` built-in constants. `FPacc` is a place where a float value can be stored temporarily. The program variables `a` and `b` also appear in the `.data` section.

Also note the `nop` instruction at the beginning of the assembly code, at start. This is necessary for `gdb` to operate correctly and allow us single stepping the assembly code.

We assembled and link edited the assembly code file `test8a.S` by the commands:

```
as -g -o test8a.o test8a.S
ld -o test8a test8a.o builtina.o
```

The `-g` switch will include enough information in the assembled code for the `gdb` to work correctly and access the assembly source code. `builtina.o` is the object code file of our built-in functions, with which we should link our program. As mentioned previously, you can use the standard C library `libc.so` with a `miniC` program if you want to.

Now we are ready to check the executable program `test8a` with `gdb`. We give the following command:

```
gdb test8a
```

Assuming that the source `test8a.S` is in the same directory, `gdb` will load both the executable and the source and display a rather verbose starting prompt. We can give simply a `run` command and `gdb` will try to execute our program, catching any exceptions and if so, stop and display them. Let us try that first.

```
gdb test8a

GNU gdb (GDB) 7.1
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
```

```
This GDB was configured as "i486-slackware-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from
/home/hbd/compiler_book/book/progs/miniC/AST/test8a...done.
(gdb)
```

Now we give the `run` command:

```
Starting program: /home/hbd/compiler_book/book/progs/miniC/AST/test8a
-2.22220E + 00
Program exited with code 014.
(gdb)
```

This shows that the program terminated normally. The exit code 014 shown is the value in CPU register `%ebx` and is irrelevant here. Note the program output −2.22220E + 00 also printed by `gdb`, as it will print all the outputs (and wait for inputs).

However, our interest here is to demonstrate the single stepping, with monitoring of floating-point unit actions. For that we have to stop the normal, full-speed execution of our program by `gdb`, by inserting a `break-point`. We do that by first deciding where we want to put the break-point. For that we see the listing first:

```
(gdb) li
1  .text
2  .globl _start
3  _start: nop
4  flds FC1
5  fstps a
6  flds FC2
7  fstps b
8  flds a
9  flds b
10  fmulp
(gdb)
```

As we want to trace the FPU actions right from the beginning, we put the break-point at line number 4 and give `run` command again:

```
(gdb) b 4
Breakpoint 1 at 0x8048075: file test8a.S, line 4.
(gdb) run
Starting program: /home/hbd/compiler_book/book/progs/miniC/AST/test8a

Breakpoint 1, _start() at test8a.S:4
4   flds FC1
(gdb)
```

The program execution is suspended at the break-point and `gdb` shows the next instruction that will be executed if we proceed further. We shall do that by the `single-stepping` command `s`, but before we do that, let us have a look at the FPU, by giving `info float` command:

```
(gdb) info float
  R7: Empty 0x00000000000000000000
  R6: Empty 0x00000000000000000000
  R5: Empty 0x00000000000000000000
  R4: Empty 0x00000000000000000000
  R3: Empty 0x00000000000000000000
  R2: Empty 0x00000000000000000000
  R1: Empty 0x00000000000000000000
=>R0: Empty 0x00000000000000000000

Status Word:         0x0000
                       TOP: 0
Control Word:        0x037f     IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0xffff
Instruction Pointer: 0x00:0x00000000
Operand Pointer:     0x00:0x00000000
Opcode:              0x0000
(gdb)
```

The gdb shows a snap-shot of the FPU. The eight floating-point registers are shown numbered R0 to R7, with the status of all of them being "empty". An arrow => indicates the Top-of-Stack. The Status Word includes flags C3, C2 and C0, which are modified according to the result of a floating-point instruction. Other information like Control Word, Tag Word, Instruction Pointer, Operand Pointer and Opcode is also shown. The display shows that the FPU is in initialized condition. Although these information is very useful in a more detailed debugging, we shall from now onwards show only the content of the eight FP registers.

Now we alternately single-step and display FPU state:

```
(gdb) s
5     fstps a
(gdb) info float
=>R7: Valid 0x3fff8e38860000000000 + 1.111099958419799805
  R6: Empty 0x00000000000000000000
  R5: Empty 0x00000000000000000000
  R4: Empty 0x00000000000000000000
  R3: Empty 0x00000000000000000000
  R2: Empty 0x00000000000000000000
  R1: Empty 0x00000000000000000000
  R0: Empty 0x00000000000000000000
```

Note that the previous display instruction, `flds FC1`, is executed, as a result of which TOS gets the value 1.111… (note the single-precision approximation) and status is "Valid". We take one more single-step and display FPU:

```
(gdb) s
6     flds FC2
(gdb) info float
```

```
  R7: Empty 0x3fff8e38860000000000
  R6: Empty 0x00000000000000000000
  R5: Empty 0x00000000000000000000
  R4: Empty 0x00000000000000000000
  R3: Empty 0x00000000000000000000
  R2: Empty 0x00000000000000000000
  R1: Empty 0x00000000000000000000
=>R0: Empty 0x00000000000000000000
```

The executed instruction `fstps a` stores the value in variable `a` and pops the FPU stack. The next two instructions – `flds FC2` and `fstps b` – do similar things for the variable `b`.

We then execute the two instructions `flds a` and `flds b` which will load the two values on the stack:

```
(gdb) s
10 fmulp
(gdb) info float
  R7: Valid 0x3fff8e38860000000000  +1.111099958419799805
=>R6: Valid 0xc0008000000000000000  -2
  R5: Empty 0x00000000000000000000
  R4: Empty 0x00000000000000000000
  R3: Empty 0x00000000000000000000
  R2: Empty 0x00000000000000000000
  R1: Empty 0x00000000000000000000
  R0: Empty 0x00000000000000000000
```

The next instruction to be executed is `fmulp` which multiplies the top two numbers and pops one value:

```
(gdb) s
11    fstps FPacc
(gdb) info float
=>R7: Valid 0xc0008e38860000000000  -2.222199916839599609
  R6: Empty 0xc0008000000000000000
  R5: Empty 0x00000000000000000000
  R4: Empty 0x00000000000000000000
  R3: Empty 0x00000000000000000000
  R2: Empty 0x00000000000000000000
  R1: Empty 0x00000000000000000000
  R0: Empty 0x00000000000000000000
```

On single stepping, we have the following FPU state:

```
(gdb) s
12    pushl FPacc
(gdb) info float
  R7: Empty 0xc0008e38860000000000
  R6: Empty 0xc0008000000000000000
  R5: Empty 0x00000000000000000000
```

```
  R4: Empty 0x00000000000000000000
  R3: Empty 0x00000000000000000000
  R2: Empty 0x00000000000000000000
  R1: Empty 0x00000000000000000000
=>R0: Empty 0x00000000000000000000
```

Note that the stack is now empty. The next few instructions will invoke the `printr` function to print the value in `FPacc`. We shall not try to trace them.

While developing FPU-based functions, it is generally advisable at the end of the function, to leave the FPU stack in the same condition as at start. For a program, it is almost mandatory to leave the FPU stack empty at the end of the program.

The above example should have helped you appreciate the considerable amount of help that `gdb` is able to give to a code developer.

12.11 Difference Between AT&T and Intel Assembly Syntax

The syntax for assembly language used by us is known as the AT&T syntax. It is the one supported by the GNU tool chain that comes as a standard with every Linux distribution. However, the official syntax for x86 assembly language (known as the Intel syntax) is different. It is the same assembly language for the same platform, but it looks different. Some of the major differences between the two syntaxes are as follows. In Intel syntax:

- Registers are not prefixed with the percent sign (%).
- A dollar sign ($) is not required to do immediate-mode addressing. Instead, non-immediate addressing is accomplished by surrounding the address with brackets ([]).
- The instruction name does not include the size of data being moved. If that is ambiguous, it is explicitly stated as BYTE, WORD or DWORD immediately after the instruction name.
- The way that memory addresses are represented is much different (shown below).
- Because the x86 processor line originally started out as a 16-bit processor, most literature about x86 processors refer to “words” as 16-bit values, and call 32-bit values “double words”. However, we use the term “word” to refer to the standard register size on a processor, which is 32-bits on a modern x86 processor. The syntax also keeps this naming convention – DWORD stands for “double word” in Intel syntax and is used for standard-sized registers, which we would call simply a “word”.
- Intel assembly language has the ability to address memory as a segment/offset pair. We did not mention this because Linux does not support segmented memory, rather it uses a flat memory model and is therefore irrelevant to normal Linux programming.
- Other differences are of rather minor nature.

To show some of the differences, consider the following instruction:

```
movl %eax, 8(%ebx,%edi,4)
```

In Intel syntax, this would be written as:

```
mov [8 + %ebx + 4 * edi], eax
```

The memory reference is a bit easier to read than its AT&T counterpart because it spells out exactly how the address will be computed. However, the order of operands in Intel syntax can be confusing.

EXERCISES

Extend the miniC compiler discussed and developed in this chapter as follows:

1. Add all the `float` operations.
2. Add array implementation – grammar rules, semantics and code generation.
3. Add structures implementation – grammar rules, semantics and code generation.
4. Add file I/O. The macros available in `macros.S` may be helpful.
5. Replace the BST Symbol Table functions in `symbol.c` by those for AVL or Red–Black trees.
6. Implement some of the Optimization techniques for `miniC`.

FURTHER READING AND WEB RESOURCES

1. "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference", Intel Corporation, 1999.
   `http://developer.intel.com/design/pentium4/manuals/245471.htm.`
2. GNU assembler: `http://www.gnu.org/software/binutils/manual/gas-2.9.1/as.html.`
3. GNU linker: `http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html.`
4. Intel "System Programming Guide":
   (`http://developer.intel.com/design/pentium4/manuals/245470.htm`), and
   (`http://developer.intel.com/design/pentium4/manuals/245472.htm`).
5. "Intel 64 and IA-32 Architectures Optimization Reference Manual" order no. 248966.

This page is intentionally left blank.

APPENDIX A

Formal Languages and Automata

In this appendix, we discuss essential portions of Formal Languages and Theory of Automata. The other names for these topics are "Theory of Computation" and "Mathematical Theory of Computer Science".

The topics discussed concern basic mathematical properties of computer hardware, software and some applications. Though they have purely mathematical and philosophical aspects, we discuss the topics from viewpoint of their connections with computer science and specifically, language processors.

For example, study of *grammars* (e.g. Section A.3) will be useful to those trying to design a new programming language for some special application. For those concerned with pattern matching, ideas covered under *Regular Expressions* and *Finite-state Machines* will be useful.

A.1 Essential Mathematical Background

In this section, we consider the mathematical background needed generally in computer science and specifically in study of Formal Languages and Theory of Automata, for those students who do not have it or who want to revise.

Why do we need mathematics at all in computer science?

- To present information in an easily assimilated form;
- To provide a convenient method for solving a problem;
- To predict the behaviour of a real system.

In short, we use mathematics to model the real-world, see Fig. A.1.

A.1.1 Formal Logic: A Language for Mathematics

In many situations, we have to represent and deal with statements describing some real-world fact or event. We should be able to express such information and process it precisely. A branch of mathematics, **Formal Logic**, is concerned with this requirement. There are several areas in logic.

- Classical logic
  - Propositional logic
  - Predicate logic

- Non-classical logic
  - Modal logic
  - Multi-valued logic
  - Fuzzy logic, etc.

We are concerned here with Propositional and Predicate logic.

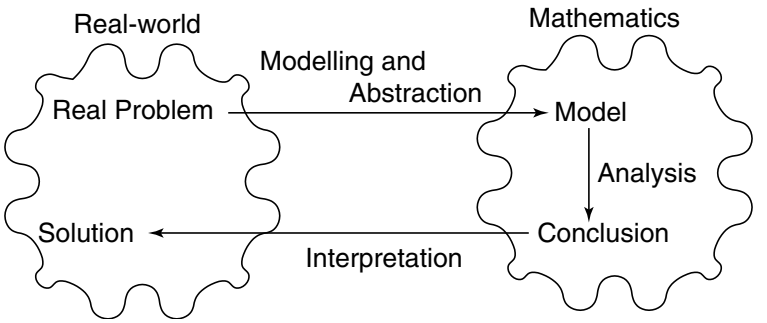


Fig. A.1 Mathematics used for modelling

Table A.1 Examples: assertions

Assertion	Assertion?	Proposition?
3 is not an even integer	Yes	True
4 is a prime number	Yes	False
The square of an integer is non-negative	Yes	True
The moon is made of blue cheese	Yes	False
x is greater than 5	Yes	No
This statement is false	Yes	No
Is he a CS student?	No	No
How are you?	No	No
What a difficult problem it is	No	No
Write this program	No	No

A.1.2 Assertions and Propositions

First, we take some definitions:

Assertion: A statement (i.e., a declarative sentence).

Proposition: An assertion which is either true or false, but not both.

Truth value: Whether a proposition is true or not.

Note that whether a proposition is true is not necessarily known. For example, "A meteorite landed on the Chowpati, Mumbai, in 258 BC" is a proposition.

In order to deal with propositions systematically and study their structures, propositions are often denoted by symbols, say P, Q, R, etc., which are called propositional variables. Note that such variables take *truth values* – **True** or **False**. A truth table of a proposition P including k propositional variables is a table which shows the truth value of P for each combination of truth values of the propositional variables.

A.1.3 Logical Connectives

The real-world situations almost always need to be expressed as a combination of facts. In propositional logic, this is modelled by a compound proposition.

A *compound proposition* is constructed from existing propositions by applying operators called **logical connectives**.

Negation – NOT operation

Notations: NOT $\neg$ ~
Definition (given by a truth table):

P	NOT P
T	F
F	T

T and F denote True and False, respectively.

This operation is used to show absence of some condition or fact. For example, if P denotes "today is Monday", then $\neg P$ denotes "today is not Monday" or "today is other than Monday".

Conjunction – AND operation

Notations: AND $\wedge$.
Definition:

P	Q	$\boldsymbol{P \wedge Q}$
T	T	T
T	F	F
F	T	F
F	F	F

This connective denotes that both of the two propositions are True.

Disjunction (Inclusive) OR Operation

Notations: OR $\vee$ +
Definition:

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

This connective denotes that either one or both of the two propositions are True.

Examples

Gopal is at school and either Mani is on the phone or Sita is at the store.

Let G denotes "Gopal is at school", M denotes "Mani is on the phone", and S denotes "Sita is at the store".

The above compound proposition is denoted by $G \wedge (M \vee S)$. Note that parentheses are used to avoid ambiguity in expressions.

Then, $(\neg M) \wedge (\neg (G \vee S))$ denote the following: Mani is not on the phone and it is not the case that either Gopal is at school or Sita is at the store.

Exclusive OR

Notations: XOR $\oplus$ ^
Definition: Note that $P \oplus Q$ has the same truth table as $(P \vee Q) \wedge \neg (P \wedge Q)$.
This connective denotes that one and only one of the two propositions is True.

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T
F	F	F

Conditional or Implication

Notations: $\rightarrow$
Definition:

P	Q	$P \rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Note that $P \rightarrow Q$ has the same truth table as $(\neg P) \vee Q$.

The left-hand side of $\rightarrow$ is called a **hypothesis** or **premise** and the right-hand side is called a **conclusion** or **consequence**.

$Q \rightarrow P$ is called the **converse** of $P \rightarrow Q$. $\neg P \rightarrow \neg Q$ is called the **contra-positive** of $P \rightarrow Q$.

Biconditional or Equivalence

Notations: $\leftrightarrow$ $\Leftrightarrow$
Definition:

P	Q	$P \leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

Note that $P \leftrightarrow Q$ has the same truth table as $(P \rightarrow Q) \wedge (Q \rightarrow P)$.

A conditional statement $P \rightarrow Q$ may correspond to the following in a natural language:

If P, then Q;

P only if Q;

P implies Q;

When P, Q;

P only when Q;

P is a sufficient condition for Q;

Q is a necessary condition for P;

Q if P;

Q follows from P;

Q, provided P;

Q is a logical consequence of P;

Q whenever P; etc.

A bidirectional statement $P \leftrightarrow Q$ may correspond to the following in a natural language:

P if and only if Q;

P is equivalent to Q;

Q is equivalent to P;

P is a necessary and sufficient condition for Q;

Q is a necessary and sufficient condition for P.

Tautologies, Contradictions and Contingencies

Tautology A proposition which is always true.

Example: $(P \rightarrow Q) \rightarrow ((\neg Q) \rightarrow (\neg P))$

In the truth table of a tautology, every row has T.

Contradiction A proposition which is always false.

Example: $(P \rightarrow Q) \wedge (P \rightarrow (\neg Q)) \wedge P$

In the truth table of a contradiction, every row has F.

Contingency A proposition which is neither a tautology nor a contradiction.

Example: $(P \rightarrow Q) \rightarrow (Q \rightarrow P)$

In the truth table of a contingency, some rows have T and the others have F.

Implementation Notes

Most programming languages, including C, provide for expressing predicates in the form of IF (condition) THEN statement-T ELSE statement-F. It is important to note that though seemingly such a construct implements $p \Leftrightarrow q$ *if and only if*, it is really $p \Rightarrow q$, i.e. *if p then q*, because q may be true or false before execution of a statement "IF(p) THEN q", but afterwards, q will be true if p is true. If you want the if-and-only-if behaviour, you must use "IF(p) THEN q ELSE not q".

One very important facility in C related to predicates is the `assert()` function, which checks any given assertion and terminates the program if it is not true, giving an error diagnostic. This facility seems to have escaped the attention of most of academic institutions, but it is important from viewpoint of debugging real-life programs.

A.1.4 Sets

A **set** is a collection of entities that can be identified. The entities are called **members** or **elements** of the set. Sets are characterized by the concept of **membership**, denoted by the symbol $\in$.

$$a \in S, \text{ the entity } a \text{ is a member of the set } S.$$

$$a \notin S, \text{ the entity } a \text{ is not a member of } S.$$

A set can be **Finite**, for example, the set of non-negative integers less than 5.

A small finite set is often written out fully as, for example $\{1, 2, 3, 4\}$.

A set can be **Empty**, for example, a set of all human beings aged more than 500 years. Such a set is also called null-set and denoted by symbol $\varnothing$.

A set can be **Infinite**, for example, the set of all C programs. This is not a finite set, since no matter how large a program you write, it is possible to write a larger one by inserting another statement.

A set can have other sets as its elements, for example, $S = \{1, 2, \{a, b\}\}$. Here, note that element a is *not* a member of set S, but rather member of anonymous set $\{a, b\}$, which is member of S.

Notations for Sets

Usually, two notations are used to specify sets:

- **Enumeration:** for example $\{0, 1, 2, 3, 4\}$
- **Set-builder form;** or, Predicate form:

  $\{x \mid P(x)\}$ where $P(x)$ is a predicate that describes a property of all elements x

  $\{x \in S \mid P(x)\}$ equivalent to $\{x \mid x \in S \wedge P(x)\}$

  For example, $\{x \mid x \text{ is an integer} \wedge 0 \leq x \wedge x < 5\}$

There are a few sets used so often that they are called **Special Sets**:

$\mathbb{N}$: The set of natural numbers (usually including 0)

$\mathbb{Z}$: The set of integers

$\mathbb{Q}$: The set of rational numbers

$\mathbb{R}$: The set of real numbers

$\mathbb{C}$: The set of complex numbers

$\mathbb{Z}_+$: Also $\mathbb{P}$, the set of positive integers

$\mathbb{Z}_n$: The set of non-negative integers less than n

$\mathbb{R}_+$: The set of positive real numbers

Relationships Between Sets

Let X and Y be arbitrary sets.

X is a **subset** of Y, denoted as $X \subset Y$, iff $\forall\, x\, (x \in X) \rightarrow (x \in Y)$ holds.

Y is a **superset** of X iff X is a subset of Y.

X and Y are **equal**, denoted as $X = Y$, iff $X \subset Y$ and $Y \subset X$.

$X \neq Y$ denotes that X and Y are not equal.

X is a **proper subset** of Y iff $X \subset Y$ and $X \neq Y$.

X and Y are said to be (mutually) **disjoint** iff no element is in both X and Y.

Set Operations

Let U denote the **universal** set that is the common set from which elements of sets are selected. The following operations on sets are defined:

Union: $A \cup B = \{x \mid x \in A \vee x \in B\}$

Intersection: $A \cap B = \{x \mid x \in A \wedge x \in B\}$

Complement: $A = \{x \in U \mid \neg (x \in A)\}$

Set difference: $A - B = \{x \mid x \text{ in } A \wedge \neg (x \in B)\}$. Note that $A - B$ is not equal to $B - A$ in general.

Cartesian product: $A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$. Elements in a Cartesian product are called **ordered tuples**.

Power set: For a set A, $2^A = \{B \mid B \subset A\}$. This is the set of all subsets of A. Some textbooks denote the power set by $P(A)$.

Notation: The union of a number of sets is denoted by

$$\bigcup_{i=1}^{n} A_i = A_1 \bigcup A_2 \bigcup \ldots A_n$$

Similarly, intersection of a number of sets is denoted by

$$\bigcap_{i=1}^{n} A_i = A_1 \bigcap A_2 \bigcap \ldots A_n$$

Properties of Sets

Sets obey the following algebraic laws:

Here, U denotes the universal set.

- Idempotent law: $A \cup A = A,\ A \cap A = A$
- Associative law: $(A \cap B) \cap C = A \cap (B \cap C),\ (A \cup B) \cup C = A \cup (B \cup C)$
- Commutative law: $A \cap B = B \cap A,\ A \cup B = B \cup A$
- Distributive law: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C),\ A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- Absorptive law: $A \cap (A \cup B) = A,\ A \cup (A \cap B) = A$
- Identity law: $A \cap U = A,\ A \cup \varnothing = A,\ A \cap \varnothing = \varnothing,\ A \cup U = U$
- Complement law: $A \cap \overline{A} = \varnothing,\ A \cup \overline{A} = U,\ \overline{A} = A,\ \overline{U} = \varnothing,\ \overline{\varnothing} = \cup$
- DeMorgan's law: $\overline{(A \cap B)} = \overline{A} \cup \overline{B},\quad \overline{(A \cup B)} = \overline{A} \cap \overline{B}$

Sets – Implementation Notes

Theory of sets forms bases of many designs – both hardware and software in computer field. They are used extensively in design of language translators. How do we represent sets in a computer? It depends upon the nature of elements in the set, size of the set and what operations we are going to perform on them in our application.

A set can be represented as:

- an array – limited size, same type of elements, fast operations;
- a linked list – arbitrary size, slow operations;
- a bit field – most useful for limited size, very fast operations;
- a hash table – arbitrary size, reasonably fast operations.

All the four methods can be used in C language, but bit-field method is the fastest and most economical in memory if large sets are to be handled. If the sets we want to handle are all of size 32 or less, then we can use an unsigned integer to represent it, each bit position representing inclusion of a particular element in the set. For example, suppose our universal set has 32 elements $\{1, 2, \ldots, 32\}$. Let set $A = \{1, 3, 6\}$ and set $B = \{3, 4, 5, 6\}$. A will be represented as $0\ldots0100101 = 00000025$ (hex), and B as $0\ldots0111100 = 0000003C$ (hex).

Union, intersection, negation and Ex-OR are very easy with this representation.

Union: Use logical OR operation, '|', e.g. $A \cup B = 00000025 \mid 0000003C = 0000003D \rightarrow \{1, 3, 4, 5, 6\}$.

Intersection: Use logical AND operation, '&', e.g. $A \cap B = 00000025$ '&' $0000003C = 00000024 \rightarrow \{3, 6\}$.

Negation: Just take complement by '~' operation.

Ex-OR: Use logical Ex-OR operation, '$\wedge$'.

Existence of a member: To check if member number i is in the set, use the expression (set & $(1 \ll (i - 1))$).

One limitation of this method is that it is difficult for cases where we have to handle mixed sets, like set of sets.

Can we use this technique for larger set sizes? Yes, for details see p. 690 [Hol90]. Elsewhere we have given a number of example programs for checking of grammars etc., which uses a different implementation for sets and set operations, for the reason indicated above. In anticipation of that development, we give here our implementation of a set. See C code files `set.h` and `set.c`.

Each member of a set is represented by a data structure shown in Fig. A.2 and a set is represented by a data structure shown in Fig. A.3. A number of processing functions like union, intersection, etc. are provided.

In Perl and Python, it is easiest to use Hashes or Dictionaries for implementing sets. Simplest representation would be:

```
$myset{able}  = 1;
$myset{baker} = 1;
```

i.e., just add hash members with element names as *key* and '1' as *value*. If more information for each element is needed to be stored, then *references* can be used in place of '1' as the value.

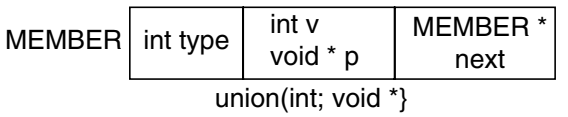


Fig. A.2 A member of a set. `type` can be INT, STRING, SET, LIST, EMPTY, DUMMY. Note the union

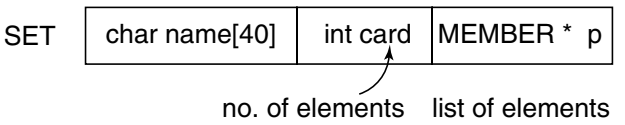


Fig. A.3 A set consists of a set header and a singly linked list

Functions

Let X and Y be sets.

A **partial function** f from X to Y, denoted $f: X \to Y$, is a subset of $X \times Y$ (i.e., the Cartesian product of X and Y) which satisfies the condition that for each $x \in X$, there is at most one $y \in Y$ such that $(x, y) \in f$.

X is called the **domain** of f and Y is called the **range** (or codomain) of f.

Notation: $f(x) = y$ iff $(x, y) \in f$.

x is called an **argument** of f and y is the **value** of f for x.

$f(x)$ is undefined if there is no y such that $(x, y) \in f$.

A partial function $f: X \to Y$ is called a total function (or simply function) if $\forall\, x \in X$, there is one and only one $y \in Y$ such that $f(x) = y$.

Operations on Functions

Let $g: X \to Y$ and $f: Y \to Z$ be functions. The **composition** of f and g, written as $f \circ g$, is a function from X to Z such that $\forall\, x \in X$, $(f \circ g)(x) = f(g(x))$.

Composition of functions is associative. If f, g and h are functions, then $(f \circ g) \circ h\ (= f \circ (g \circ h)$.

The **inverse** of $f: X \to Y$ is a function $g: Y \to X$ such that $\forall\, x \in X$, $(g \circ f)(x) = x$.

Note that the inverse of f does not necessarily exist.

Notation: f^{-1} denotes the inverse of f if it exists.

Classification of Total Functions

Surjection $f: X \to Y$ is **surjective** (or **onto**) iff $f(X) = Y$.
Injection $f: X \to Y$ is **injective** (or **one-to-one**) iff $x \neq x'$ implies $f(x) \neq f(x')$. This means that if $f(x) = f(x')$, then $x = x'$.
Bijection $f: X \to Y$ is **bijective** (or **one-to-one correspondence**) iff f is both surjective and injective.

Theorem A.1.1 *A function f is bijective iff f^{-1} exists.*

Cardinality of a Set

Let S be a set and let $\mathbb{Z}_n = \{0, 1, 2, 3, \ldots, n - 1\}$.

S is a **finite** set of **cardinality** n iff there is a bijection $f: \mathbb{Z}_n \to S$. We write $|S| = n$.

S is **infinite** iff S is not finite.

S is **countably infinite** iff there is a bijection $f: N \to S$, where N is the set of natural numbers 0, 1, 2, …. The cardinality of a countably infinite set is called $\aleph_0$, pronounced "aleph zero".

S is said to be **countable** if S is finite or countably infinite. S is said to be **uncountably infinite** (or **uncountable**) if there is no bijection $f: N \to S$.

Theorem A.1.2 *The set R of real numbers is uncountably infinite.*

This theorem is usually proved by a process known as **Diagonalization**. For further details, see Section A.1.9.

Two sets A and B are said to have the same cardinality if there is a bijection between A and B.

If there is a bijection $f: R \to S$, where R is the set of real numbers, the cardinality of S is called $\aleph_1$.

Implementation Notes

Almost all modern programming languages provide for definition and use of functions.

A.1.5 Relations

Cartesian Product (Cross Product)

$A \times B = \{(x, y) \mid x \in A \wedge b \in B\}$. Here the ordered pairs are sometimes denoted by $<x, y>$

$$A_1 \times A_2 \times \ldots \times A_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in A_i \, \forall i \, (1 \leq i \leq n)\}$$

The *ordered n-tuple* $(x_1, x_2, \ldots, x_n)$ is sometimes denoted by $< x_1, x_2, \ldots, x_n >$.

An ***n*-ary relation** R among $A_1, A_2, \ldots, A_n$ is a subset of $A_1 \times A_2 \times \ldots \times A_n$.

$R = \{(x_1, x_2, \ldots, x_n) \mid P((x_1, x_2, \ldots, x_n)) \wedge x_i \in \mathrm{A}_i \, \forall i, (1 \leq i \leq n)\}$ where P is a predicate representing properties of R. n is called the **arity** of R.

If $n = 2$, R is called a **binary relation** from A_1 to A_2. A_1 is called a **domain** and A_2 is called a **range** or **codomain**.

If $A = A_1 = A_2 = \ldots = A_n$, R is called a **relation** on A.

For example,

Let $P(x, y, z)$ denote "student x has been studying y for z years". $X =$ a set of all students; $Y =$ a set of majors; $Z =$ a set of positive integers.

$$\begin{aligned} R &= \{(x, y, z) \in X \times Y \times Z \,/\, P(x, y, z)\} \\ &= \{(\text{Alka,Math,2}),(\text{Jayant,EE,4}),(\text{Dravid,ICS,1}), \ldots\} \end{aligned}$$

Binary Relations and Digraphs

We now focus on binary relations on a set A.

Notation: (x, y) in R iff xRy, i.e. x is R-related to y

Graphical Representation

$$A = \{1, 2, 3, 4\} \; R = \{(2, 1), (2, 2), (3, 1), (3, 4), (4, 3)\}$$

an element in A is equivalent to a vertex (or node) and an element in R is equivalent to an edge (or arc).

These two sets form what is known as a *Directed graph* (*digraph* for short), denoted by $G = (V, E)$, where V is a set of vertices and E is a set of edges which are ordered pairs of vertices (see next section).

Classification of Binary Relations

Let R be a binary relation on a set A.

- R is **reflexive** iff $(x, x) \in R \; \forall \, x \in A$.
- R is **irreflexive** iff $(x, x) \notin R \; \forall \, x \in A$. Note that R can be neither reflexive nor irreflexive.
- R is **symmetric** iff $(x, y) \in R \rightarrow (y, x) \in R \; \forall$ pairs $x, y \in A$.
- R is **antisymmetric** iff $(x, y) \in R$ and (y, x) in $R \rightarrow x = y \; \forall \, x$ and $y \in A$. Note that R can be both symmetric and antisymmetric, and R can be neither symmetric nor antisymmetric.
- R is **transitive** iff $(x, y) \in R$ and $(y, z) \in R \rightarrow (x, z) \in R \; \forall$ triple of x, y and $z \in A$.

Relations Types in Digraph Representation

Reflexive: Every vertex has a self-loop.

Irreflexive: No vertex has a self-loop.

Symmetric: If there is an edge from one vertex to another, there is an edge in the opposite direction.

Antisymmetric: There is at most one edge between distinct vertices.

Transitive: If there is a path from one vertex to another, there is an edge from the vertex to another.

Operations on Relations

Because a binary relation R is just a set of pairs drawn from the basic sets A and B, the set operation like Union, Intersection, Set Difference, Complement, Inverse (or Converse) are defined on a pair R_1, R_2 of them. Further, let R_1 be a binary relation from A to B and R_2 be a binary relation from B to C. Then the operation of composition is defined on them as:

$R_1 \circ R_2 = \{(x, z) \mid \exists y, [(x, y) \in R_1 \wedge (y, z) \in R_2]\}$ which is a binary relation from A to C. This relation has properties of Associativity and Distributivity over union.

Let R be a relation on A. The ***n*-th power** R^n is *identity* relation defined as $\{(x, x) \mid x \in A\}$ if $n = 0$; and is $R^{n-1} R$ if $n > 0$.

The **Transitive Closure** of a relation is defined as:

$$R^* = \bigcup_{i \geq 0} R^i$$

Equivalence Relations and Equivalence Classes

Let R be a binary relation on A. Then R is an **equivalence relation** on A iff R is reflexive, symmetric and transitive.

For example,

Let Z be the set of integers, and n be a positive integer. Let $R = \{(i, j) \mid i \cong j \pmod{n}\}$, i.e. $R = \{(i, j) \mid \exists\, k, [kn = i - j]\}$.

Then,

1. R is reflexive, since $i \cong i \pmod{n}\ \forall\, i \in Z$.
2. R is symmetric, since $i \cong j \pmod{n}$ implies $j \cong i \pmod{n}$.
3. R is transitive.

Partitions

Let X be a set of subsets $S_1, S_2, \ldots, S_m$ of A.

X is called a **partition** of A iff

1. $A = \bigcup_{i=1}^{m} S_i$ and
2. $S_i \cap S_j = \varnothing\ \forall\ 1 \leq i \leq m,\ 1 \leq j \leq m$ and $i \neq j$, m is called the rank of X.

Order Relations

An **order relation** is a transitive binary relation on a set A. A **partial order** on A is a reflexive, antisymmetric, transitive binary relation on A. The ordered pair (A, R) is called a partially ordered set or **poset**. Elements x and y in A are said to be **comparable** iff either xRy or yRx holds.

A partial order R on A is called a **total order** on A iff for every pair x and y in A, x and y are comparable.

The ordered pair (A, R) is called a **totally ordered set**.

For example,
The relation "less than or equal to" on the set of natural numbers:

- **Reflexive:** For every $i \in N$, $i \leq i$ holds.
- **Antisymmetric:** For every i and j ($i \neq j$), exactly one of $i < j$ and $j < i$ holds.
- **Transitive:** If $i \leq j$ and $j \leq k$ hold, then $i \leq k$ holds.

Thus, $\leq$ is a partial order on N.

A.1.6 Inductive Definitions

Recursive definition in more general is called *Inductive Definition*.

Let S be an infinite set to be defined.

An **inductive definition** of S consists of the following three components:

Base Clause (or Basis): To establish that a finite number (usually one) of certain objects are elements in the set S;

Inductive Clause (or Induction): To establish a way to obtain a new element from some of the previously defined elements of the set S;

Extremal Clause (sometimes left implicit): To assert that unless an object can be shown to be an element of the set S by applying the base and inductive clauses only a finite number of times, the object is not an element of S.

Example: Even Integers

Suppose that the universe is the set of integers. The set E of even integers can be defined inductively as follows:

1. **Base Clause:** $0 \in E$.
2. **Inductive Clause:** If $x \in E$, both of $x + 2$ and $x - 2$ are in E.
3. **Extremal Clause:** No integer is in E unless it can be shown to be so by a finite number of applications of Clauses (1) and (2).

Example: Power Set

Let A be a set. The power set 2^A (i.e., the set of all subsets of A) can be inductively defined as follows:

1. **Base Clause:** The empty set $\varnothing \in 2^A$.
2. **Inductive Clause:** If $X \in 2^A$ and $w \in A$, then the union $X \cup \{w\} \in 2^A$.
3. **Extremal Clause:** Nothing else in 2^A other than sets obtained by applying (1) and (2) a finite number of times.

A.1.7 Proof Techniques

First, we consider some definitions:

Axiom: An assertion that is assumed to be true.

Hypotheses: An assertion proposed as a theorem

Theorem: An assertion that can be shown to be true.

Proof: An argument to establish the truth of a theorem.

Inference rules: A means of deducing a theorem from axioms and/or previously proved theorems.

There is no general algorithm for deducing systematically whether a given assertion is true or false.

The construction of proofs is an art and a craft. The skill to develop a proof can only be learned by means of examples and practice.

Generally, a proof consists of a series of assertions from which the final assertion is derived and that is the **Conclusion** – the assertion to be proved. For example, a typical *inference rule* is:

if $P \rightarrow Q$ and
$Q \rightarrow R$ **then**
$P \rightarrow R$ (conclusion)

There are a number of such **Inference Rules** which help us in developing a proof. Some of them are:

if P **then** $P \vee Q$, can also be written as $P \Rightarrow P \vee Q$
if $P \wedge Q$ **then** P
if P, $P \rightarrow Q$ **then** Q this rule known as Modus Ponens
if $\neg Q$, $P \rightarrow Q$ **then** $\neg P$ Modus Tollens
if $P \vee Q$, $\neg P$ **then** Q
if P, Q **then** $P \wedge Q$
if $P \rightarrow Q$, $R \rightarrow S$, $P \vee R$ **then** $Q \vee S$
if $P \rightarrow Q$, $R \rightarrow S$, $\neg Q \vee \neg S$ **then** $\neg P \vee \neg R$

Various Proof Methods

Vacuous Proof ($P \rightarrow Q$) is true if P is false.

Trivial Proof ($P \rightarrow Q$) is true if Q is true.

Direct Proof: Assume that P is true. Show that Q must be true, by applying inference rules.

Proof of the Contrapositive: Show that the contrapositive $((\neg Q) \rightarrow (\neg P))$ of $(P \rightarrow Q)$ is true, by using other proof techniques.

Proof by Contradiction: Assume that $(P \rightarrow Q)$ were false, i.e. P were true and Q were false. Derive a contradiction such as $(R \wedge (\neg R))$ from this assumption, by applying inference rules.

Reduction to Absurdity: Assume that $(P \rightarrow Q)$ were false. Derive $(\neg R)$ from this assumption for a known theorem R.

A.1.8 Proof by Induction

The concept of induction provides not only a method of defining infinite sets, but also powerful techniques for proving assertions of the form $\forall x, P(x)$, where the universe is an inductively defined set (e.g., the set of natural numbers).

A proof by induction usually consists of two parts corresponding to the base and inductive clauses of the definition of the universe S.

Base Step (or Basis) to establish $P(x)$ is true for every element x specified in the base clause of the definition of S.

Inductive Step (or Induction) to establish that $P(x)$ is true for each element x constructed in the inductive clause of the definition of S, assuming that $P(y)$ *is true for all the elements y used in the construction of x*. This assumption is called **Induction Hypothesis**.

Note that there is no step corresponding to the extremal clause of the definition of S. Since the extremal clause in the definition of S guarantees that all elements of S can be constructed using only the base and inductive clauses a finite number of times, $P(x)$ holds for all elements of S.

For example,
Let the alphabet $\Sigma = \{(,)\}$. A set B of strings over Σ is the subset of Σ^+ such that

1. **Basis:** $( ) \in B$.
2. **Induction:** If x and $y \in B$, then (x) and $xy \in B$.
3. **Extremal:** B consists of all strings over Σ which can be constructed by a finite number of applications of (1) and (2).

Theorem A.1.3 *Let L(x) denote the number of left parentheses in $x \in B$ and R(x) denote the number of right parentheses in $x \in B$. $\forall$ x, L(x) = R(x).*

Proof by Induction:

1. **Basis:** $x = ( ) \in B$, since $L(x) = 1$ and $R(x) = 1$, $L(x) = R(x)$ holds.
2. **Induction:** Assume that $L(x) = R(x)$ and $L(y) = R(y)$ for x and $y \in B$.
   Case 1: Consider (x) in B. $L((x)) = L(x) + 1$ $R((x)) = R(x) + 1$
   By induction hypothesis, $L(x) = R(x)$, $L(x) + 1 = R(x) + 1$. Hence, $L((x)) = R((x))$ holds.
   Case 2: Consider xy in B. $L(xy) = L(x) + L(y)$ $R(xy) = R(x) + R(y)$
   By induction hypothesis, $L(x) + L(y) = R(x) + R(y)$. Hence, $L(xy) = R(xy)$ holds.
   Therefore, $L(x) = R(x)$ holds for all x in B.

First Principle of Mathematical Induction

Suppose that the universe is the set of natural numbers $N = \{0, 1, 2, \ldots\}$. Let $P(n)$ be a predicate on N.

Inference Rule: Given $P(0)$; if $\forall\, n$, $P(n) \rightarrow P(n + 1)$ then $\forall\, n$, $P(n)$.

Proof Technique:

1. **Basis:** Show that $P(0)$ is true, using whatever proof technique is appropriate.
2. **Induction:** Let n be an *arbitrary* element in the universe. Assume that $P(n)$ is true (induction hypothesis). Show that $P(n + 1)$ is true.

For example,

To prove: $\forall\, n, \quad \sum_{i=0}^{n} i = \dfrac{n(n+1)}{2}$.

Here, $P(n) = \sum_{i=0}^{n} i = \dfrac{n(n+1)}{2}$.

Proof: Induction on n

1. **Basis:** $n = 0$ LHS $\sum_{i=0}^{0} i = 0$ RHS $= 0(0+1)/2 = 0$. Thus, the equality holds. Hence, $P(0)$ is true.
2. **Induction:** Assume that $P(n)$ is true, i.e. $\sum_{i=0}^{n} i = n(n+1)/2$ holds. Consider $P(n + 1)$. $\sum_{i=0}^{n+1} i = (\sum_{i=0}^{n} i) + (n+1) = n(n+1)/2 + (n+1)$ by induction hypothesis $= (n + 1)(n + 2)/2$. Hence, $P(n + 1)$ is true.

One has to be careful in applying the principle of inductive proof, otherwise a wrong proof can result.

Second Principle of Mathematical Induction

Inference Rule: If $\forall\, n$, $[\forall\, k, k < n \rightarrow P(k)] \rightarrow P(n)$ then $\forall\, n$, $P(n)$.

Proof Technique: Let n be an arbitrary element in the universe. Assume that for every $k < n$, $P(k)$ is true. Show that $P(n)$ is true.

This proof technique is more powerful than the first principle of mathematical induction.

For example,

To prove: All integers $n \geq 2$ can be written as a product of prime numbers.

Here, $P(n) =$ "n can be written as a product of primes".

Proof: Induction on $n \geq 2$

Assume that $\forall\ k$ such that $2 \leq k < n$, k can be written as a product of primes. We will show that n can be written as a product of primes. The proof is by cases.

Case 1: n is a prime. Then n is such a product of one prime that is itself.

Case 2: n is not a prime. Then n must have a factor i, where $2 \leq i \leq n$. Thus, we can write $n = ij$, where $2 \leq j < n$. By the induction hypothesis, both i and j can be written as products of primes. Thus, n can be written as a product of the products of primes.

In both cases, $P(n)$ is true.

Note that this cannot be proved by the first principle of mathematical induction.

Proof by Induction is used in showing the validity of **Recursive** algorithms like Factorial or Tower of Hanoi.

Execution Trace and Recursive Functions

We apply inductive proof technique to find the *Execution Trace* (ET) of any algorithm or program in general and a recursive function in particular.

Execution Trace: It is the exact order in which statements which make up an algorithm or a program are executed (for a given input data.)

An algorithm or a program has two aspects – the static or lexical structure, that is what we see when it is written down, and a dynamic structure. The execution trace denotes the dynamic structure, what happens when the program is executed.

The concept of ET is important in theory of formal languages and also in debugging (removing errors) and profiling (finding out time taken by various components) of programs.

Problem: Given a pseudo-C function:

```
rec(n){
    if(n == 0) S0;
    else{
        A(n);
        rec(n-1);
        B(n);
    }
}
```

Give an inductive proof that for a function call `rec(i)`, $i \geq 1$, the order of execution of the statements `S0`, `A(i)`, `B(i)`, etc. will be:

```
A(i), A(i-1), ... A(1), S0, B(1), B(2), ... B(i-1), B(i)
```

Proof:

Basis: For $i = 0$, the sequence is `S0`, obvious from the definition of `rec()`.

Hypothesis: For $i > 0$, the sequence is `A(i), A(i-1), ... A(1), S0, B(1), B(2), ... B(i-1), B(i).`

Induction: For $i + 1$, from the definition of `rec()`, the sequence is `A(i+1)`, `sequence for i`, `B(i+1)`. Proved.

Note that statement/s which come before the recursive call are executed in reverse order or order of recursive descent, while statement/s which are after the recursive call are executed in order of ascent from recursion.

In anticipation of a later section on regular expressions, Section A.4, this ET can be expressed by a regular expression `A*SB*`.

Structural Induction

A special form of induction is sometimes useful. Let U be a set and set $I \subseteq U$. Let O be a set of operations on the elements of U. Now we define a subset L of U as follows:

1. $I \subseteq L$;
2. L is closed under each of the operations in O;
3. L is the smallest set satisfying (1) and (2).

Then to prove that every element of the set L has some property P, it is sufficient to prove the following two things:

1. Every element in I has property P;
2. The set of elements of L having property P is closed under each of the operations in O.

Example

Given language $L \subseteq \{a, b\}^*$ defined as:

1. $a \in L$;
2. $\forall x \in L$, $ax \in L$;
3. $\forall x, y \in L$, strings bxy, xby, $xyb \in L$;
4. no other strings except as defined by (1), (2) and (3) are in L.

Here, $I = \{a\}$, $O = \{a.x, b.xy, x.b.y, xy.b\}$ and (4) ensures that this is the smallest such language. Now, we want to prove that "every string in L has more a's than b's". We prove that as:

1. for I: a has more a's than b's (obvious);
2. for closure under $O - 1$: $\forall x \in L$ having more a's than b's, ax also has (obvious);
3. for closure under $O - 2$: $\forall x, y \in L$ having more a's than b's, each of the strings $b.xy$, $x.b.y$, $xy.b$ also have (xy will have at least two a's more than b's, so by adding a b does not change the property.)

A.1.9 Cantor's Theory of Counting and Infinite Sets

From the viewpoint of their sizes, there are three kinds of sets:

- finite
- infinite:
  - countably infinite, and
  - uncountable.

The nineteenth century mathematician George Cantor showed that only correct way to compare sizes of two sets is by one-to-one correspondence of their elements. This idea becomes more important when we are dealing with infinite sets.

For example, consider two sets – set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ and the set of even numbers $\mathbb{E} = \{0, 2, 4, \ldots\}$. Both are infinite sets. From our common-sense thinking, we would argue that as even numbers are included in natural numbers, size of set $\mathbb{E}$ should be smaller than size of $\mathbb{N}$. Cantor said this is not so, because as you compare the two sets by one-to-one correspondence of their respective elements – matching 0 in $\mathbb{N}$ with 0 in $\mathbb{E}$, 1 in $\mathbb{N}$ with 2 in $\mathbb{E}$, etc – we find that for each element of $\mathbb{N}$ there is a corresponding element in $\mathbb{E}$. Thus, both the sets are of the same size. Cantor proved that the set $\mathbb{R}$ of real numbers is *uncountable* based on this idea. He gave the proof by what is known as *diagonal argument*, which was used subsequently for proving many results in Formal Languages and Automata, specially related to *computability*.

Theorem A.1.4 *The set of real numbers [0,1) = {x ∈ ℝ / 0 ≤ x < 1} is uncountable.*

Proof: Proof by contradiction. Note that a real number in [0,1) has infinite decimal expansion and no integer part. We shall disallow decimal expansions ending in infinite number of 9's. Then every real number has exactly one decimal representation.

Suppose the set [0,1) is countable, i.e. $[0,1) = \{a_0, a_1, a_2, \ldots\}$. For each $i \geq 0$, let the decimal representation be

$$a_i = .a_{i,0}a_{i,1}a_{i,2}\ldots$$

Now we construct a number $x \in [0,1)$ which is not in the assumed set $\{a_0, a_1, a_2, \ldots\}$. We do this by:

$$x_i = \begin{cases} a_{i,i} + 1 & \text{if } a_{i,i} < 8 \\ 7 & \text{if } a_{i,i} \geq 8 \end{cases}$$

and the number $x = .x_0x_1x_2\ldots$. Note that we have defined x in terms of the *diagonal* of an infinite matrix:

$$\begin{matrix} \mathbf{a_{0,0}} & a_{0,1} & a_{0,2} & \cdots \\ a_{1,0} & \mathbf{a_{1,1}} & a_{1,2} & \cdots \\ a_{2,0} & a_{2,1} & \mathbf{a_{2,2}} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

This gives the name to the method of the proof.

It is clear that $\forall\, i \geq 0, x \neq a_i$. We assumed that $x \in [0,1)$, so this gives a contradiction. Thus, the set [0,1) is uncountable. This means that whatever method you choose to count or enumerate the elements of the set, you will always have many more elements left out *uncounted*.

Theorem A.1.5 *If S is any countably infinite set, then the set of all subsets of S, 2^s is uncountably infinite. This means that for any non-empty alphabet Σ, the set of all the languages over Σ is uncountable.*

Proof: As there is one-to-one and onto mapping (bijection) from $\mathbb{N}$ to S, there would be a bijection from $2^{\mathbb{N}}$ to 2^S. Then it is sufficient to show that $2^{\mathbb{N}}$ is uncountable.

Suppose $2^{\mathbb{N}}$ is countable and $= \{A_0, A_1, A_2, \ldots\}$. Each A_i is defined by $i \notin A_i$. We now choose a set of natural numbers $A \subseteq \mathbb{N}$, such that it differs from A_i with respect to i, $\forall i \geq 0$, i.e. $A = \{i \in \mathbb{N} \mid i \notin A_i\}$.

Thus, though $A \in 2^{\mathbb{N}}$, $A \notin \{A_0, A_1, A_2, \ldots\}$. This is a contradiction. Thus, $2^{\mathbb{N}}$ is uncountable and hence 2^s is uncountable.

Pigeonhole Principle or Dirichlet Drawer Principle: If $k + 1$ or more objects are placed into k boxes, then there must be at least one box containing two or more of the objects.

Generalized Pigeonhole Principle: If N objects are placed into k boxes, then there is at least one box containing at least $\lceil (N / k) \rceil$ objects.

For example,

During a month with 30 days, Anand plays at least 1 game of Chess a day, but not more than 45 games in the month.

There must be a period of some number of consecutive days during which he must play exactly 14 games.

Proof: Let a_i be the number of games played on or before the i-th day of the month. Then, the sequence $a_1, a_2, \ldots, a_{30}$ is a strictly increasing sequence of positive integers with $a_{30} \leq 45$. Consider another sequence obtained by adding 14 to every number in the above sequence. $(a_1) + 14$, $(a_2) + 14$, $\ldots$, $(a_{30}) + 14$. It remains to be shown that it is a strictly increasing sequence. In addition, $15 \leq (a_1) + 14$ and $(a_{30}) + 14 \leq 59$ hold.

Now, consider the concatenation of the above two sequences.

$$a_1, a_2, \ldots, a_{30}, (a_1)+14, (a_2)+14, \ldots, (a_{30})+14$$

It includes 60 integers which are between 1 and 59. Thus, by the Pigeonhole principle, there must be two integers in the concatenation such that they are equal. One of them must be in the first half and another must be in the second half. Let a_i and $(a_j) + 14$ be such integers that are equal. Then, we have $a_i - a_j = 14$. It implies that Anand plays exactly 14 games between the j-th day and the i-th day.

Theorem A.1.6 (A theorem proved by the Pigeonhole principle.) *Every sequence of $(n^2 + 1)$ distinct real numbers contains a subsequence (not necessarily consecutive) of length $n + 1$ that is either strictly increasing or strictly decreasing.*

Theorem A.1.7 *The number of surjection from X with m elements to Y with n elements ($m \geq n$) is $n^m - C(n, 1)(n - 1)^m + C(n, 2)(n - 2)^m - \ldots + (-1)^{n-1}C(n, n - 1)1^m$.*

A.2 Formal Language Theory Review

A.2.1 Strings

Let Σ be a finite, non-empty set of symbols, called an **alphabet**.

A string (or word) over Σ is a finite sequence of symbols put side-by-side.

For example, *abacc* is a string over $\{a, b, c\}$.

The length of string s over Σ, denoted by $|s|$, is the total number of occurrences of symbols in s.

For example, $|abacc| = 5$.

A string of length 0 is called the **empty** string or **null** string, denoted by ϵ or λ.

A.2.2 Binary Operation on Strings

Let $x = a_1\ a_2\ \ldots\ a_m$ and $y = b_1\ b_2\ \ldots\ b_n$ be strings over Σ, of lengths m and n respectively. The **concatenation** of x and y, denoted by xy, or $x \circ y$ is the string $a_1\ a_2\ \ldots\ a_m\ b_1\ b_2\ \ldots\ b_n$.

A.2.3 Relationships Between Strings

A string x is called a **prefix** of a string z iff $\exists\, y$, $[z = xy]$. A string x is called a **suffix** of a string z iff $\exists\, y$, $[z = yx]$. A string x is called a **substring** of a string z iff $\exists\, u\ \exists\, v\ [z = uxv]$.

x is a proper prefix of z iff x is a prefix of z and $x \neq z$. x is a proper suffix of z iff x is a suffix of z and $x \neq z$. x is a proper substring of z iff x is a substring of z and $x \neq z$.

Σ^* denotes the set of all strings over Σ.

Theorem A.2.1 *Let Σ be an alphabet. $(\Sigma^*, \circ, \epsilon)$ forms a monoid, where $\circ$ is concatenation and ϵ is the empty string.*

This theorem means that strings as defined have associative operator $\circ$ and a unique identity element ϵ.

Sometimes, especially when applying the formal language theory to programming languages, we use an alternative nomenclature.

Definition A.2.1 (Vocabulary) *Vocabulary V is a finite, non-empty set of symbols:*

$$V = v_1, v_2, \ldots$$

Definition A.2.2 (Strings) *Strings are formed by concatenation operation '.', for example, 'a.b', which can also be written as 'ab' if there is no chance of confusion between symbols 'a', 'b' and 'ab'. In short, it means zero or more symbols from some vocabulary V, placed side-by-side in a series or sequence.*

Definition A.2.3 (Length of a string) *A string has length equal to number of symbols in it and is denoted by $| s |$.*

Definition A.2.4 (Set of strings) *Let V^1 denote the set of all the strings of length 1 from V, V^2 denote the set of all the strings of length 2 from V, etc.*

Then, $V^2 = V^1.V^1$; similarly for V^3, V^4, etc.

Then $V^+ = V^1 \cup V^2 \cup \ldots$ and $V^* = \{\epsilon\} \cup V^1 \cup V^2 \cup \ldots$.

String Sets as Algebraic System

λ or ϵ is the identity element in the algebraic system: $< V^*, ., \lambda >$, because for any $s \in V^*$, $s.\lambda = \lambda.s = s$. Comparing with $< N, +, 0 >$, we find that the two systems are *isomorphic*. This result is important and forms basis of many of our later derivations.

A.2.4 Languages

In short, a *Language* L over vocabulary V is given by $L \subset V^*$. Usually, some "alphabet" V_T (a set of Terminal symbols) is used.

We shall call the strings in a language a *sentence*. Although structurally they look similar, note that "string" and "sentence" carry different meanings. A string is any arbitrary member of the set V^*, while a sentence has to be a member of a particular language L, which may be defined over V^*, i.e. $L \subset V^*$. A language has some form of discipline built into it, as specified by its grammar (see below).

We have already seen that Σ^* denotes the set of all strings over Σ.

Inductive Definition of Σ^*

1. **Base Clause:** ϵ is in Σ^*.
2. **Inductive Clause:** If x is in Σ^* and $b \in \Sigma$, then $xb \in \Sigma^*$.
3. **Extremal Clause:** Σ^* includes only those strings which can be constructed by a finite number of applications of (1) and (2).

A **language** over Σ is a subset of Σ^*.
For example, let $\Sigma = \{a, b\}$.
Language $L = \{a^n b^n \mid n \geq 0\} = \lambda, ab, aabb, aaabbb, \ldots$

A.2.5 Binary Operation on Languages

Let X and Y be languages over Σ. The **concatenation** of X and Y, denoted by XY, is defined as $XY = \{xy \mid x \in X \text{ and } y \in Y\}$.

For example,

$$X = \{a, ab, bb, baa\},\ Y = \{0, 11\} \text{ and } XY = \{a0, a11, ab0, ab11, bb0, bb11, baa0, baa11\}$$

Note that XY and YX are different in general.

A.2.6 Power X^i of a Language X

Inductive definition of X^i of a language X is:

1. **Base Clause:** $X^0 = \{\epsilon\}$
2. **Inductive Clause:** $X^{i+1} = (X^i)\, X$

For example,

$$Y = \{0, 11\},\ Y^2 = Y^1\ \{0,11\} = (Y^0\ \{0, 11\})\{0, 11\}) = (\{\lambda\}\{0, 11\})\{0, 11\} = \{00, 011, 110, 1111\}$$

A.2.7 Closure of a Language

Let X be a language over Σ. The **closure** of X, denoted by X^*, is defined as follows:

$$X^* = \bigcup_{i \geq 0} X^i = X^0 \cup X^1 \cup X^2 \cup \ldots$$

The positive closure of X, denoted by X^+, is defined as follows:

$$X^+ = \bigcup_{i \geq 1} X^i = X^1 \cup X^2 \cup \ldots$$

Note that the set of all strings over Σ is equal to the closure of Σ, where Σ is viewed as a set of strings of length 1 rather than an alphabet. This is the reason we denote the set of all strings over Σ by Σ^*.

We often regard X^* as a unary operation to X, called **Kleene star**.

A.3 Grammars

A correct specification of a programming language requires:

1. Specification of a set of symbols, called *Terminals*;
2. The set of all syntactically correct programs;
3. The "meaning" of such programs.

Possible methods in which we can specify a language:

1. Exhaustive listing of each and every sentence in that language.
2. Use a *Generative* method: give rules by which correct sentences can be formed. The set of such rules, which must be finite, is called a *Grammar*.

   For certain restricted class of languages, known as Regular Languages, a simpler generative method, known as *Regular Expression*, which is a kind of formula, is generally used (see Section A.4).
3. Use a machine or *Acceptor* which, when fed with any string $s \in V^*$, will tell us "Accepted" or "Rejected", depending upon whether $s \in L$ (see Fig. A.4).

Fig. A.4 An acceptor – if it says 'Yes' the string is in the language, if 'No' then it is not

The rules in a grammar are called *Production* rules or simply Productions.

Meta-language: It is a system or language used to describe precisely another language. Indian grammarian Panini was the first inventor of such a language.

In the present era, grammars were studied extensively by **Noam Chomsky** and others, while Indian Sanskrit grammarian **Panini** gave a highly systematic grammar some 2500 years back in his book Ashtadhyayi. The method used by him to specify the grammar – a meta-language – was quite similar to what we use (Backus Normal Form) today.

The three-way relationship between a Language, its Grammar and corresponding Acceptor is shown in Fig. A.5.

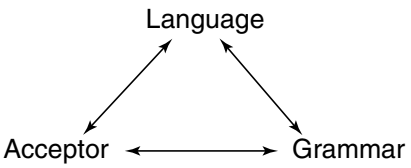


Fig. A.5 Relationship between a language, grammar and acceptor

A grammar can be used in two ways:

1. Generate a sentence, e.g. a programmer uses the grammar of a programming language while writing her programs to ensure that they strictly belong to the language.
2. Design the Acceptor, i.e. a device to check sentences in the language, a Compiler.

We shall mainly study two kinds of Formal Languages – Regular Expressions and Context-free Languages – in terms of their grammars, with an intention of finding algorithms to design the Acceptors for them. As it is shown in Chapters 3 and 4 on the front-end of compilers, we can utilize those algorithms to design the corresponding phases of a compiler.

A typical programming language is a three-layered language:

1. At lexical level, the constructs can be expressed in terms of regular expressions.
2. At expression level and nested control structure level, the constructs can be expressed in terms of CFL.
3. Simple control structures themselves can be expressed in terms of regular expressions.

Example

Consider a grammar:

```
G0:
      I -> L
      I -> IL
      I -> ID
```

```
L -> a
L -> b
...
D -> 0
D -> 1
...
```

This grammar generates sentences which are identifiers in a typical programming language, like abc, asl34, xc56d, etc. In other words, it specifies and generates a language of identifiers.

The right arrow symbol denotes a production or *replacement.* It means that the symbol on the LHS can be replaced by the string on the RHS. Apart from the right arrow, there are two kinds of symbols used – the capital letters denote *non-terminals* or *meta-symbols* and the small letters and digits denote *Terminals* or *final-symbols.*

Generally, the symbol on the LHS of the first rule is called the *starting* symbol. When we wish to generate a sentence in our language, we start with a string which contains only the starting symbol. We then successively use various applicable rules in the grammar (in any order we choose) to replace all the non-terminals (NT) with Terminals (T). We stop when no more replacement can take place by use of any of the grammar rules.

For example, we can generate a sentence 'abl23d' like this:

$$\text{I} \rightarrow \text{IL} \rightarrow \text{IDL} \rightarrow \text{IDDL} \rightarrow \text{IDDDL} \rightarrow \text{ILDDDL} \rightarrow \text{LLDDDL} \rightarrow \text{aLDDDL} \rightarrow \text{abDDDL} \rightarrow \text{ab1DDL} \rightarrow \text{ab12DL} \rightarrow \text{ab123L} \rightarrow \text{ab123d done.}$$

Note that we had to think a bit about which of the rules we shall apply at each stage of the production. Had we chosen a rule wrongly, we would not have got the required sentence.

Note that the above grammar is defined in a *Recursive* way. This is fundamental to definition of infinite languages.

Formal Grammars

A formal grammar is defined as:

$$G = <V_N, V_T, S, P>$$

where
V_N = set of non-Terminals,
V_T = set of Terminals,
S = the starting symbol, $S \in V_N$ P = productions, a set of rules. P has the general form:

$$(V_N \cup V_T)^* V_N (V_N \cup V_T)^* \rightarrow (V_N \cup V_T)^*$$

Note that V_N is a set of Terminal symbols in the *meta-language* in which the grammar is written.

Direct Derivative

Let G be a grammar and $s, t \in V^*$, where $V = V_N \cup V_T$.

s is called a *direct derivative* of t, written as $t \Rightarrow s$, if there are strings s_1 and s_2 in V^*, such that $t = s_1\ \alpha\ s_2$, $s = s_1\ \beta\ s_2$, and $\alpha \rightarrow \beta$ is a production (rule) in P.

s is called *derivative*, $t \Rightarrow_+ s$, if there are strings $s_0, s_1, s_2, s_3, \ldots$ in V^*, such that $t = s_0$, $s_0 \Rightarrow s_1$, $s_1 \Rightarrow s_2$, etc. and $s_n = s$.

Then, a language $L = \{s \mid S \Rightarrow_+ s, \text{ and } s \in V_T^*\}$.

A.3.1 Methods of Grammar Specification

Ways of specifying a grammar:

1. Backus Normal Form (BNF) and modified or extended BNF;
2. syntax or "Railroad" diagrams;
3. syntax trees.

Note that a Chomsky type-3 language (Regular Language) can also be specified by a *formula* known as Regular Expressions, see Sections A.3.2 and A.4.

Backus Normal Form

The grammar G0 in the above example can be expressed in BNF as:

```
<identifier>  ::= <letter> | <identifier> <letter> |
                  <identifier> <digit>
<letter>      ::= 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
```

Extended BNF

The BNF is extended by additional constructs:
[*e*] – 0 or 1 occurrence of *e*,
{*e*} – 0 or more occurrences of *e*,
{*e*}*m*, *n* – minimum *m* and maximum *n* occurrences of *e*,
'*c*' – Terminal symbol *c*.

Railroad Diagrams

Railroad diagrams are pictorial means for specifying a grammar and are most useful to a programmer as a reference for the syntax of the programming language he plans to use (see Fig. A.6). The arrows

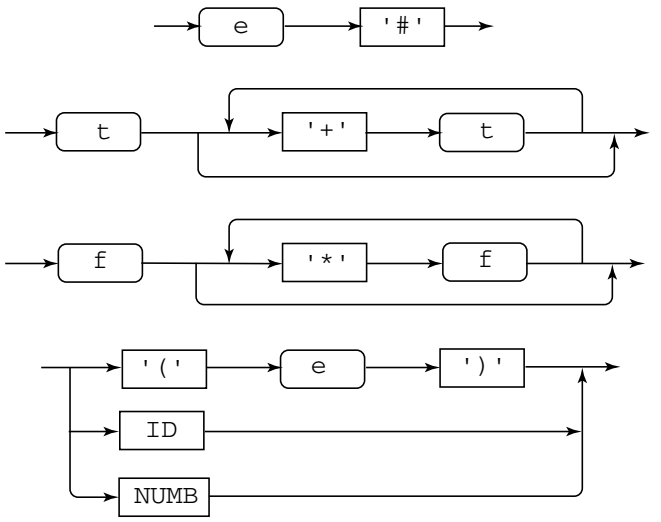


Fig. A.6 Railroad diagrams: for top – a statement, a simple expression using only addition of terms, a term and a factor in an arithmetic expression. Compare with corresponding grammar

show direction in which the expansion or replacement takes place. Expansion starts from the LHS and proceeds generally towards RHS.

Syntax Trees

A grammar can also be represented by an *abstract syntax tree.* For example, the example grammar of an arithmetic expression is shown in Fig. A.7.

In this tree, a node is either a NT symbol, a bullet (•) or a T symbol. The interpretation of the tree is as follows:

NT The branches denote alternative productions; if it is a leaf-node, then it is subject to further replacements by the sub-tree under it.

• Denotes concatenation operation of its branches.

T The Terminal symbol stands for itself in the productions.

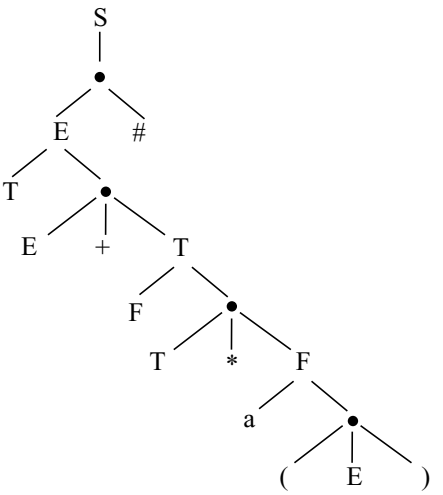


Fig. A.7 An abstract syntax tree for example grammar

A.3.2 Chomsky Classification

During his study or languages, Noam Chomsky classified languages – Natural, programming and Formal – into four classes, as shown in Table A.2.

Examples and Exercises

1. $L = \{a^n b^n c^n \mid n \geq 1\}$

```
P = { 1. S -> aSBC
2. S -> abC
3. bB -> bb
```

Table A.2 Classification of languages as per Noam Chomsky

Chomsky type	Name	Acceptor	Grammar characteristics
0	Free or unrestricted	… …	No restrictions
	Recursively enumerable	Turing machine	Phrase-structured Gr
1	Context-sensitive	LBA	$\alpha \rightarrow \beta$ $\mid \alpha \mid \leq \mid \beta \mid$
2	Context-free	PDA	$\alpha \rightarrow \beta$ $\mid \alpha \mid \leq \mid \beta \mid$ and $\alpha \in V_N$
3	Regular	FSM	$\alpha \rightarrow \beta$ $\mid \alpha \mid \leq \mid \beta \mid$, and $\alpha \in V_N$ and $\beta = Ba$ or a (or aB)

Note: LBA – Linear Bounded Automata, PDA – Push-down Automata, FSM – Finite-state Machine (Finite Automata)

```
4. bC -> bc
5. CB -> BC
6. cC -> cc
}
```

Confirm that the grammar does generate the language specified.

2. $L = \{a^n bc^n \mid n \geq 1\}$

```
P = { 1. S -> a C c
2. C -> a C c
3. C -> b
}
```

Confirm that the grammar does generate the language specified.

3. Grammar G1:

```
P = { 1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> a
}
```

What language does the grammar generate?

4.
```
P = { 1. S -> aS
2. S -> aB
3. B -> bC
4. C -> aC
5. C -> a
}
```

What language does the grammar generate?

5. Grammar G2:

```
P = { 1. V -> N
2. V -> +N
3. V -> -N
4. N -> 0 | 1 | ... |9
5. N -> 0N | 1N | ... | 9N
}
```

What language does the grammar generate?

6.
```
P = { 1. S -> iEtS
2. S -> iEtSeS
3. S -> a
4. E -> b
}
```

What language does the grammar generate? What kind of language is it?

7. Grammar G3 $= < \{V, S, R, N\}, \{+, -, ., d, \#\}, V, P >$

```
P = { 1. V -> SR#
2. S -> +
3. S -> -
4. S -> e
5. R -> .dN
6. R -> dN.N
7. N -> dN
8. N -> e
}
```

What language does the grammar generate? What kind of language is it?

A.4 Regular Languages, Regular Expressions and Finite-state Machine

Regular languages (RL) are specified by a Regular Expression (RE), a Regular Grammar (RG) or indirectly by its acceptor, a Finite-state Machine (FSM).

A.4.1 Regular Languages

Three basic set operations of union, intersection and closure are used to define a new language from an existing one. If we start with a finite vocabulary V_T and build a language using only these operations, we get a Regular Language and the formula by which it is specified is called Regular Expression.

Recursive Definition of RL

A RE denotes a RL, which is also called its *valuation* or meaning, see Table A.3.

A language L over V_T is an RL if there is an RE over V_T which specifies it.

We sometimes write r^n to denote $\underbrace{rr\ldots r}_{n \text{ times}}$, and r^+ to denote rr^* or r^*r.

Table A.3 Recursive definition of a regular expression and regular language

	RE	Valuation (meaning or language)
1	Φ is a RE	Empty language Φ,
2	λ is a RE	$\{\lambda\}$ – A single null string
3	$\forall a \in V_T$ a is a RE	Language $\{a\}$
4	$\forall$ *RE* r, s *over* V_T, each of the following is RE:	
	rs	$L_r L_s$, where L_i is valuation of i
	$r + s$	$L_r \cup L_s$
	r^*	L_r^*
	(r)	Used for only controlling precedence
5	Only those expressions defined by rules 1 to 4 only from RE over V_T	

Example

Write an RE to specify a language over {0, 1}, having at least one 1.

The possible RE are: (0)*1(0 + 1)*, (0 + 1)*1(0 + 1)* and (0 + 1)*1(0)*. Each of them emphasis a 1 in different positions.

Example

$V_T = \{+, -, ., E, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,

Let RE $S = \lambda + \text{'+'} + \text{'}-\text{'}$

RE $d = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$

Then consider an RE $Sd^+ (d^+ + .d^+ESd^+ + ESd^+)$ – this is the format of real constants in many programming languages.

Theorem A.4.1 *Every finite language is regular.*

You can prove this by induction over the number of sentences in the language. You will also have to prove a lemma that any finite length string can be expressed as an RE.

Regular operations: Let L_1 and L_2 be languages. The following *regular operations* are defined:

Union: $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$

Concatenation: $L_1 \circ L_2 = L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$

Star: $L_1^* = \{x_1x_2\ldots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$

Note that an empty string λ is always a member of any L^*.

We state the following theorem without proving it.

Theorem A.4.2 *The class of Regular Languages is closed under the regular operations: union, concatenation and star.*

A.4.2 Finite Automaton or Finite-state Machine

A Finite-state Machine (FSM) or Finite Automaton (FA) is a model for a computational process which uses a *limited* and *pre-specified* amount of memory. Although we use state-transition diagram for understanding FSM and specification small FSMs, a formal, mathematical definition is more precise and can be used for any large-sized FSM.

An FSM consists of a finite number of *states*, an input alphabet, a starting state, one or more final states and a state-transition function which specifies rules by which the "machine" goes from one state to another.

An FA is formally defined as a 5-tuple $M = < Q, V_T, M, q_0, F >$:

where, Q = a finite set of states,

V_T = a finite set – input vocabulary,

q_0 = the *start* state,

$F \in Q$ = set of *final* or *accept* states, and

M is *transition function* or mappings of the form:

$$Q \times V_T \rightarrow Q'$$

Here, Q' is a set related to the set Q. If $Q' = Q$, then we have a deterministic FSM or DFSM. In that case, for each input symbol, only one arrow emanates from a state. That means the action taken by the machine for each input symbol is uniquely defined. We shall see other possible values of Q' later.

In order to do some computation, an FSM starts in the starting state, goes from one state to another as per the current input symbol and at the end of input string it can be in either one of the final states (the string accepted) or any other state (string rejected.)

Deterministic FSM

A deterministic FSM (DFSM) is a 5-tuple $M = < Q, V_T, M, q_0, F >$
where, Q = a finite set of states,
V_T = a finite set – input vocabulary,
q_0 = the start state,
$F \in Q$ = set of final states, and
M is mappings of the form:

$$Q \times V_T \rightarrow Q$$

It is usual to give the transition function in the form of a table.

Example

Figure A.8 shows a very simple FSM.

A formal definition of $M_2 = < \{q_1, q_2\}, \{0, 1\}, q_1, \{q_2\}, M >$, where $M =$

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

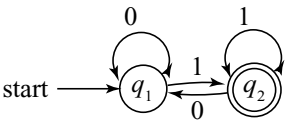


Fig. A.8 State diagram of an FSM M_2

What is the language accepted by M_2? We notice that all arrows leading to the state q_2 are labelled as '1'; thus, a string will take the machine to the final state if it ends in a '1'. Thus, $L(M_2) = \{x \mid x \in 0, 1\}^+$ and x ends in a 1}.

If we specify another machine M_2', with q_1 as the final state instead of q_2, other things being the same, then $L(M_2') = \{x \mid x$ is λ or $\in \{0, 1\}^+$ and x ends in a 0}.

Computation Done by an FSM

We now define formally and precisely the computation done by an FSM.

Let an FSM $M = < Q, \Sigma, q_0, F, \delta >$, and let $x = x_1x_2x_3 \ldots x_n$ be a string over alphabet Σ. Then M *accepts* x if a sequence of states $s_0, s_1, s_2, \ldots, s_n \exists$ in Q with the conditions:

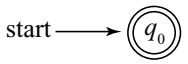
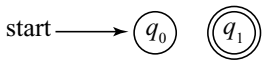
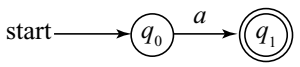
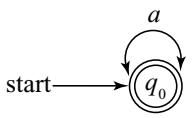
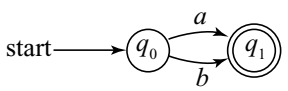
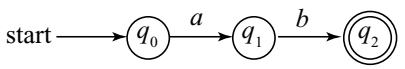
1. $s_0 = q_0$
2. $\delta(s_i, x_{i+1}) = s_{i+1}$, for $i = 0,1, \ldots, n$
3. $s_n \in F$

Thus, M *recognizes* a language L iff $L = \{x \mid M$ *accepts* $x\}$.

Definition A.4.1 *A language is called a* ***Regular Language*** *iff some finite-state machine recognizes it.*

Table A.4 shows relationships between primitive FSMs and basic operations in REs. These ideas will be our starting point while designing an FSM for accepting a given RE.

Table A.4 Relation between FSM and basic operations in RE

FSM	RE	Remarks
	λ	Any input, empty RE
	*null*	No RE, final-state not reachable
	**a**	*a* RE
	**a***	0 or more of RE *a*
	**a|b**	Alternation, either RE *a* or *b*
	**ab**	Catenation RE *a* followed by *b*

A.4.3 Non-deterministic FSM

Till now we have talked about deterministic FSM (DFSM), where at every step of computation by an FSM, the next state was uniquely determined. Although for most of the actual application of FSM theory we would definitely like to work in terms of DFSMs, the concept of non-determinism is very useful in the process of deriving such a machine. It also had considerable influence on the theory of automata. A non-deterministic FSM (NDFSM, NFA) is a generalization of DFSM and thus every DFSM (DFA) is also an NDFSM (NFA).

A small NDFSM is shown in Fig. A.9.

An NDFSM has one or more of the following generalizations:

- Multiple next states for some current states and input symbols. In the example NDFSM in Fig. A.9, we have multiple transitions for '0' at state q_1.
- "Spontaneous" transitions, i.e. transition from one state to another *without* any input. For example, in Fig. A.9, the machine can go from state q_1 to q_2 without any input (and of course, with input '0'). Such spontaneous transitions are denoted by ϵ symbol.

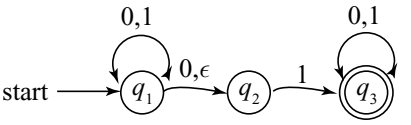


Fig. A.9 An NDFSM. Note the multiple transitions for '0' at state q_1 and "spontaneous" transition denoted by ϵ

How do we interpret the working of such a machine? There are three approaches we can take. One way is to think that the decision to select which of the multiple transitions is to be taken, or if a "Spontaneous" transition is to be taken or not, is determined by an agency external to the NDFSM. As if we have a *wizard* to tell the machine what to do in such a situation. In fact, in actual applications of FSMs, such external means are often available.

The second way is to consider that the decision is taken randomly and we observe the behaviour of the machines over many trials with such random decisions at each state where the non-determinism is present.

The third way is to consider that at every state where non-determinism is present – either multiple transitions or "Spontaneous" transition – a copy of the machine for each of the possible decisions is created and all such machines work in parallel to complete the computation. Thus, in our example machine in Fig. A.9, two extra machines are created at state q_1, one will take path to q_2 on '0' input, second will go straight to q_2 without waiting for any input. The original machine will remain in state q_1 for both '0' and '1' inputs.

Non-deterministic FSM

An NDFSM is defined as a 5-tuple $M = < Q, V_T, M, q_0, F >$, where the mapping M is $Q \times V_T \rightarrow 2^Q$, i.e. one or more next states per transition. There is a variant of NDFSM, called NDFSM with ϵ-rules, which has the mapping function M of the form $Q \times (V_T \cup \{\epsilon\}) \rightarrow 2^Q$.

Theorem A.4.3 *Every NDFSM has an equivalent DFSM.*

Two machines are *equivalent* if they recognize exactly the same language.

Conversion of an NDFSM to DFSM

We illustrate the procedure by an example. Consider the NDFSM shown in Fig. A.10, which we want to convert to an equivalent DFSM.

For M_4, $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b, \epsilon\}$, starting state and final state is q_1. We shall call the equivalent DFSM D_4.

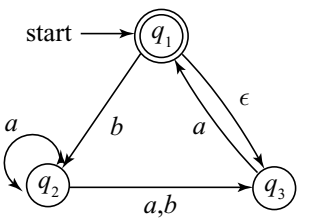


Fig. A.10 An example NDFSM, M_4

1. Determine the states of D_4: As M_4 has three states, D_4 will have $2^3 = 8$ states, one for each of the subsets of $\{q_1, q_2, q_3\}$. So the states are $\{\varnothing, \{q_1\}, \{q_2\}, \{q_3\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_2, q_3\}, \{q_1, q_2, q_3\}\}$.
2. Determine Start state of D_4: This is q_1 plus any states reachable by ϵ from q_1, i.e. q_3. Thus, start state $= \{q_1, q_3\}$.
3. Determine final states of D_4: All states of D_4 which *contain* the final states of M_4. Thus, final states of D_4 is the set $\{\{q_1\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_2, q_3\}\}$.
4. Determine the transition function for D_4: At each of the states of D_4, it goes to one state for 'a' and one state for 'b'. For example, because in M_4 there is a transition from q_2 to itself and to q_3, in D_4 there will be a transition from $\{q_2\}$ to $\{q_2, q_3\}$. We prepare a table showing all possible transitions:

D_4 state	a	b	Final D_4 state
$\varnothing$	$\varnothing$	$\varnothing$	A
$\{q_1\}$	$\varnothing$	$\{q_2\}$	
$\{q_2\}$	$\{q_2, q_3\}$	$\{q_3\}$	B

$\{q_3\}$	$\{q_1, q_3\}$	$\varnothing$	C
$\{q_1, q_2\}$	$\{q_2, q_3\}$	$\{q_2, q_3\}$	
$\{q_1, q_3\}$	$\{q_1, q_3\}$	$\{q_2\}$	D
$\{q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_3\}$	E
$\{q_1, q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$	F

As states $\{q_1\}$ and $\{q_1, q_2\}$ have no transitions going to them, they can be removed from the final D_4 and it will have the six states A, B, C, D, E, F, as shown.

A.4.4 Conversion of a Regular Grammar to FSM

We have already seen how we can convert an RE to an FSM (see Table A.4). Now we consider how can we convert a regular grammar to an FSM. Suppose we are given a grammar $G = < V_N, V_T, S, P >$, we want $M = < Q, V_T, q_0, F, M >$, which will accept exactly the language generated by G. We can do this by the following steps:

- States Q correspond to $V_N \cup \{X\}$, where $X \notin V_N$ denotes final state;
- State q_0 corresponds to S;
- States F corresponds to X;
- Construct the mapping M as:
  1. If there is a rule $A \rightarrow aB$ in P, then B in $M(A, a)$;
  2. If there is a rule $A \rightarrow a$ in P, then X in $M(A, a)$.
- We usually end up with an NDFSM. We have already seen a method of converting NDFSM to DFSM above in Section A.4.3.

Regular Grammar

A grammar $G = < V, \Sigma, S, P >$ is *regular* iff every production in the set P has one of the following two forms:

1. $B \rightarrow aC$
2. $B \rightarrow a$

where B and C are NT and a is a T symbol.

Theorem A.4.4 *For any language $L \subseteq \Sigma^*$, L is regular iff there is a regular grammar G, such that $L(G) = L - \{\lambda\}$.*

A.4.5 Languages Not in the Class Regular

Consider the language $L_1 = \{0^n\ 1^n \mid n \geq 0\}$. This is not a regular language because the acceptor for it would require arbitrary number of states, in its design. It will have to remember how many '0's occurred before start of the sequence of '1'. Note that once an FSM is designed, you cannot change the number of states in it. If you provide for say 1000 states to remember input, it will be good for $n \leq 1000$ only.

Similarly, $L_2 = \{x \mid x$ has equal number of 0's and 1's$\}$ is non-regular.

Next, consider $L_3 = \{x \mid x$ has equal number of '01' and '10'as substrings$\}$. It seems to be a non-regular but is actually a regular language. How can we decide about such languages if they are regular?

Pumping Lemma for Regular Languages

All regular languages have a special property, described in terms of a theorem known as *Pumping Lemma.* This theorem in effect says that all strings in an RL can be "pumped" (i.e., inflated like a balloon), if they have certain minimum length, known as *pumping length.* This means that every string of sufficient length in an RL, has a substring which can be repeated many times to get new strings which are still in the language.

A FSM will have a "loop" in its transition diagram, if it is an acceptor for an infinite language. (Remember, finite languages are any way regular.)

Theorem A.4.5 *If L is an infinite regular language, then there is a number p, where if $s \in L$ and $|s| \geq p$, then s may be divided into three parts, uvw, satisfying:*

1. $\forall\, m \geq 0,\ uv^m w \in L$,
2. $|v| > 0$ *and*
3. $|uv| \leq p$.

See Fig. A.11.

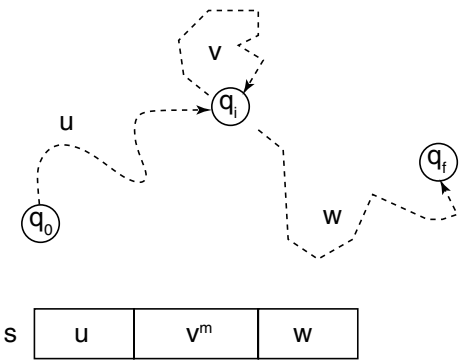


Fig. A.11 String *s* made up of *u*, $v^m w$ related to the transition diagram

Proof: Let $M = <Q, \Sigma, q_0, F, \delta>$ be a DFSM recognizing a language L and let p equal to the number of states, $|Q|$.

Let $s = s_1 s_2 \ldots s_n$ be a string in L, of length n, $n \geq p$. Let $q_0 q_1 q_2 \ldots q_n$ be the sequence of states encountered while processing this string s by M, such that $q_{i+1} = \delta(q_i, s_{i+1})$, for $0 \leq i \leq n$. This sequence has a length of $(n + 1) \geq p + 1$. Then by Pigeonhole principle, two states in the first $p + 1$ states in the sequence must be the same. Let these be q_j and q_k in the sequence. Then, $k \leq (p + 1)$. Let $u = s_1 s_2 \ldots s_{j-1}$, $v = s_j \ldots s_{k-1}$ and $w = s_k \ldots s_n$. As states q_j and q_k are the same in the machine, just as M accepts $s = uvw$, q_n being one of the final states, M has to accept all $uv^m w$ for $m \geq 0$. As $j \neq k$, $|v| > 0$ and $k \leq (p + 1)$ $|uv| \leq p$.

Example

Using the pumping lemma, show that the language $L = \{a^n b a^{2n} \mid n \geq 0\}$ is not regular.

We have to show that the pumping lemma fails, whatever be the choice of u, v, w, so if we can find even one combination of u, v, w satisfying the lemma, then the language is regular. Select $uv = a^n b$, $v = b$. Then, $w = a^{2n}$. Then, $uv^m w \notin L$. Note that if at all PL has to hold, v has to be selected of the form $a^i b a^j$, where $i \leq n$ and $j \leq 2n$. For any of the selections, PL fails.

Example

Show that $L_3 = \{x \mid x$ has equal number of '01' and '10' as substrings$\}$ is regular. As stated above, if we can find even one combination of u, v, w satisfying the lemma, then the language is regular. String $101 \in L_3$ but 1010 is not. See Fig. A.12 for an implementation of an FSM recognizing L_3.

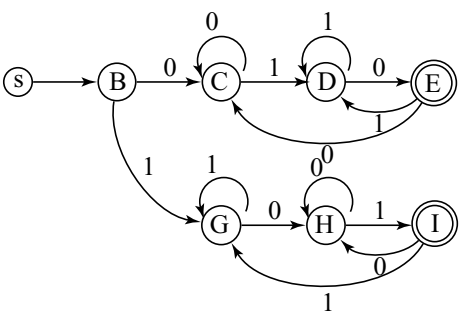


Fig. A.12 An FSM recognizing language L_3

Consider string '0010' $\in L_3$. Here, we can take $u = 0$, $v = 0$, $w = 10$ and see that PL is satisfied. You can try with other strings accepted by the FSM and confirm that L_3 is indeed regular. In fact, the very fact that we are able to give an FSM for it shows that it is regular.

A.4.6 Implementation Notes

REs are used extensively in computer science and engineering at several levels and places. Apart from its use in design of language compiler, assemblers, etc. they are also used in:

- Command language of Unix/Linux like languages;
- In utilities like `grep, sed, awk`;
- In text editors, pagers like `less`;
- Perl has extensive facilities for handling RE;
- Java, C and C++ libraries have functions for handling RE;
- It has been used heavily in decoding genetic code sequences;
- RE-to-C code generators like `lex` and `flex` are used in language translator development;
- RE and FSM are used in design of ASICs and FPGA-based systems.

A.5 Context-free Languages, CFG and Push-down Automata

Context-free Languages (CFL) have structures which cannot be specified by regular expressions. Most of the computer programming languages are of this type. The grammar which can specify a CFL is called Context-free Grammar (CFG) and acceptor for such a language is called Push-down Automata (PDA).

A.5.1 Context-free Language

We consider first an example:

Palindrome language PAL over $\Sigma = \{a, b\}$ can be defined as:

1. $\lambda, a, b \in PAL$
2. $\forall\, S \in PAL, aSa, bSb \in PAL$
3. no other string in PAL unless it can be obtained by applying (1) and (2).

Typical strings in PAL are: $a, b, aa, bb, aaa, aba, bab, \underbrace{aa\ldots a}_{\text{prefix}}\, b\, \underbrace{aa\ldots a}_{\text{rev(prefix)}}, \ldots$

We can write the corresponding grammar immediately as:

1. $S \rightarrow \lambda \mid a \mid b$
2. $S \rightarrow aSa \mid bSb$

Note the occurrence of S on the RHS in rule (2), which makes it a recursive production. This is one of the characteristic features of a CFG.

A.5.2 Context-free Grammars

A CFG is a 4-tuple $G =< V_N, \Sigma, S, P >$, where
V_N is a finite set of variables or *non-terminals*;
Σ is a finite set, alphabet, of finals or *Terminals*; $V_N \cap \Sigma = \varnothing$;
S is the starting symbol, $S \in V_N$;
P is a finite set of *Production Rules*, of the form $A \rightarrow \alpha$, where $A \in V_N$ and $\alpha \in V^* = (V_N \cup \Sigma)^*$.

We use the symbol $\underset{G}{\Rightarrow}$, for example $\alpha \underset{G}{\Rightarrow} \beta$ to mean "β is derivable from α" which denotes that a string (a *sentential form*) β can be derived from α, using a rule of grammar G.

The closure operation $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ means "β is derivable from α in zero or more steps".

Note the use of the three words:

String: Any arbitrary string derived from an alphabet;

Sentence: Any string strictly in a specified language, i.e. derivable from staring symbol S, containing only T symbols;

Sentential form: A string of T and NT symbols derivable from the staring symbol S.

We also use the following terms while discussing parsing theory:

Phrase: In a sentential form, $\sigma = \phi_1 \beta \phi_2$, β is a *phrase* if $S \overset{*}{\Rightarrow} \phi_1 A \phi_2$, $A \in V_N$ and $A \overset{+}{\Rightarrow} \beta$.

Simple phrase: If in the above formula, $A \Rightarrow \beta$, then β is a *simple phrase*.

Handle: Of a sentential form is, it is leftmost simple phrase.

Definition: Let $G = < V_N, \Sigma, S, P >$ be a CFG. The language generated by

$$G \text{ is } L(G) = \left\{ x \in \sum^* \middle| S \overset{*}{\underset{G}{\Rightarrow}} x \right\}.$$

A given language L is CFL if there is a grammar CFG G such that $L = L(G)$.

We give here a theorem about CFL, without giving its proof.

Theorem A.5.1 (Union, catenation and closure of CFL) *If L_1 and L_2 are CFL, then so are $L_1 \cup L_2$, $L_1 L_2$ and L_1^*.*

Derivation and Ambiguity

Example: Consider a grammar for algebraic expressions, G_1: $V_N = \{E\}$, $\Sigma = \{+, -, *, /, (,), a\}$, $S = E$, and P is given by:

$P = \{$
1. $E \rightarrow E + E$
2. $E \rightarrow E - E$
3. $E \rightarrow E * E$
4. $E \rightarrow E/E$
5. $E \rightarrow (E)$
6. $E \rightarrow a$

$\}$

This is an *ambiguous* grammar, because there is more than one way in which a sentence can be derived, see Fig. A.13.

To remove this ambiguity, G_1 can be rewritten as a non-ambiguous grammar or the ambiguity can be *resolved* by some other means, such as defining the *precedence* of the *operator* symbols.

A normalized or *canonical* derivation is used to remove even a trivial ambiguity where the resultant derivation tree is the same.

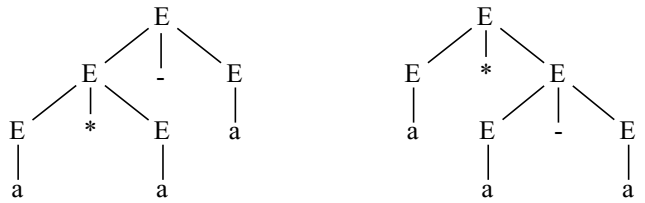


Fig. A.13 Two possible *derivation* trees for *a* * *a* − *a*

Definition A.5.1 (Leftmost derivation) *A leftmost derivation is one in which, at every step of derivation from some initial NT to the final sentence, we always replace the leftmost NT in the sentential form.*

We could also similarly define a *rightmost* derivation which is used in *Bottom-up* parsing.

Definition A.5.2 (Ambiguous grammar) *A CFG G is ambiguous if at least one sentence in L(G) has more than one distinct leftmost derivations.*

A well-known example of ambiguity is the nested `if-then-else` structure.

A non-ambiguous version for G_1, G_1': $V_N = \{E, T, F\}$, $\Sigma = \{+, -, *, /, (,), a\}$, $S = E$, and P is given by:

$P = \{$
1. $E \rightarrow E + T \mid E - T \mid T$
3. $T \rightarrow T * F \mid T/F \mid F$
3. $F \rightarrow (E) \mid a$
$\}$

Simplification of Grammar

There are several features in a grammar which make it difficult to deal with:

λ-productions have the form $A \rightarrow \lambda$. If $\lambda \in L(G)$, we cannot remove all such productions, but can ensure that $L(G) - \{\lambda\}$ can be generated by a λ-less grammar.

Unit productions of the form $A \rightarrow B$ or even $A \rightarrow A$.

Useless NT and productions which are never used in production of any sentence in the language. An NT may be either *unreachable* – no path from *S*, or *dead-end* – no further productions.

Chomsky Normal Form

A grammar given in this form is not only simplified, but has further restrictions on the productions.

A CFG $G = < V_N, \Sigma, S, P >$ is in CNF if every production in P is in one of the forms $A \rightarrow BC$, or $A \rightarrow a$, where $A, B, C \in V_N$ and $a \in \Sigma$.

Greibach Normal Form

A grammar given in this form is useful for constructing parsers.

A CFG $G = < V_N, \Sigma, S, P >$ is in GNF if every production in P is of the forms $A \rightarrow \alpha$, where α can be represented by an RE ΣV_N^*, i.e. each RHS starts with a T symbol followed by zero or more NTs.

To convert a CFG to GNF, we must remove any *left-recursive* rules of the form $A \rightarrow A\alpha$.

A.5.3 Push-down Automata

Finite-state machines are limited in ability due to their limited "memory", which is in the form of the *states* of the FSM. As we have finite and fixed number of states in any particular FSM, it cannot recognize certain kinds of languages.

If we add a separate, possibly arbitrary size, memory to an FSM, then can we make the machine more powerful? The FSM would then keep track of only the overall situation and details of the input string, sentential forms, etc. be held in that memory. PDA is such a machine, with a push-down stack (a LIFO) as the memory (see Fig. A.14).

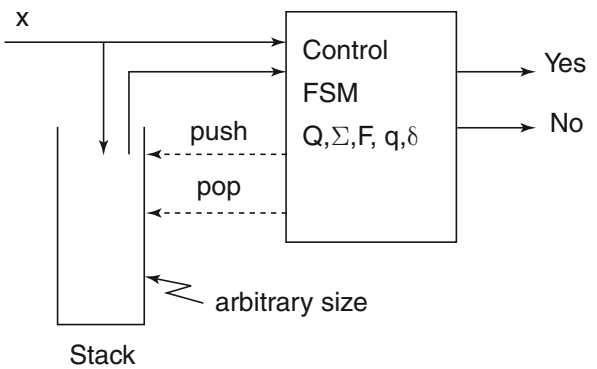


Fig. A.14 A push-down automaton. FSM part in heavy lines

Each action of a PDA depends upon the current state of the FSM controller, the next input symbol and top-of-stack symbol (TOS). The actions are:

1. Decide the next state of FSM and take a transition to that state;
2. Replace the TOS with one or more symbols, including λ, which means effectively pop-off TOS.

Definition A.5.3 (PDA) *A PDA is a 7-tuple, $M = < Q, \Sigma, \Gamma, q_0, Z_0, \delta, F >$ where*

Q – *set of states in FSM part;*
Σ – *input alphabet;*
Γ – *alphabet on the stack;*
q_0 – *the starting state of FSM;*
Z_0 – *the initial symbol on the stack;*
δ – *transition function:* $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$;
F – *set of acceptance or final states of FSM.*
If it is a non-deterministic PDA (NPDA), then δ: $Q \times \Sigma \times \Gamma \rightarrow$ *finite subsets of* $(Q \times \Gamma^*)$.

While showing the contents of the stack, it is helpful if it is shown horizontally as $\underbrace{s_0}_{\text{TOS}}\ s_1 \ldots \underbrace{s_n}_{\text{bottom}}$.

Definition A.5.4 (PDA configuration) *A configuration of PDA, written as (q, x, α), where $q \in Q$ is the current state of FSM, $x \in \Sigma^*$ unread portion of the input and $\alpha \in \Gamma^*$ is current stack contents.*

We say that some configuration $(p, ay, X\beta)$ derives some other configuration $(q, y, \gamma\beta)$, written as

$$(p, ay, X\beta) \vdash_M (q, y, \gamma\beta)$$

if δ contains the pair (q, γ).

We can extend this definition for k steps by writing $\vdash_M^k$ and similarly, $\vdash_M^*$ defines a derivation in zero or more steps.

A PDA can accept a string in two ways:

Definition A.5.5 (Acceptance by final state) *For a PDA defined as a 7-tuple as above, then a string $x \in \Sigma^*$ is accepted by final state if $(q_0, x, Z_0) \vdash_M^* (q_f, \lambda, \alpha)$ for some $\alpha \in \Sigma^*$ and $q_f \in F$.*

Definition A.5.6 (Acceptance by empty stack) *A string $x \in \Sigma^*$ is accepted by empty stack if $(q_0, x, Z_0) \vdash_M^* (q, \lambda, Z_0)$ for some $q \in Q$.*

The corresponding languages are denoted by $L_f(M)$ and $L_e(M)$.

Theorem A.5.2 *The two types of acceptances are equivalent, i.e. they recognize the same language.*

Example

Consider the language $L = \{xcREV(x) \mid x \in \{a, b\}^*\}$. Here, $\Sigma = \{a, b, c, \nabla\}$, $\Gamma = \{a, b, Z_0\}$, $Q = \{q_0, q_1, q_2\}$, q_0 is the starting state, $F = \{q_2\}$. Note that symbol ∇ is used to denote end of string marker.

The transition function δ is shown in Table A.5.

Table A.5 Transition function of an example PDA

State	TOS	*a*	*b*	*c*	∇
q_0	Z_0	(q_0, aZ_0)	(q_0, bZ_0)	(q_1, Z_0)	-
	a	(q_0, aa)	(q_0, ba)	(q_1, a)	-
	b	(q_0, ab)	(q_0, bb)	(q_1, b)	-
q_1	Z_0	-	-	-	(q_2, Z_0)
	a	(q_1, λ)	-	-	-
	b	-	(q_1, λ)	-	-

For an input string "abcba", the operation of the machine is shown Table A.6.

Table A.6 Operation of example PDA

Stack	Unread input	State
$abcba\nabla$	Z_0	q_0
$bcba\nabla$	aZ_0	q_0
$cba\nabla$	baZ_0	q_0
$ba\nabla$	baZ_0	q_1
$a\nabla$	aZ_0	q_1
∇	Z_0	q_1
	Z_0	q_2 (Accepted)

Theorem A.5.3 *For every CFG G, there is a PDA recognizing the language L(G).*

Theorem A.5.4 *For every PDA M, there is a CFG G, generating a language L(G), such that M recognizes the language L(G).*

A.5.4 Pumping Lemma for Context-free Languages

The *pumping lemma* for class CFL is derived from the corresponding grammar. Consider the grammar G_1' used as an example in Section A.5.2. If we derive a string "$a * (a + a)$" in this grammar, the derivation tree will be as shown in Fig. A.15.

This derivation tree can be broken down into three "jig-saw pieces" as shown in Fig. A.16.

The pieces are named as A, B and C and represent portions of the given string and the NT from which that portion is derived. The label at the top of a piece shows the "socket" where the piece can be plugged-in. For example, piece B can be plugged-in at a socket labelled F. So can piece C also. Some of the ways in which we can plug the pieces is shown in Fig. A.17.

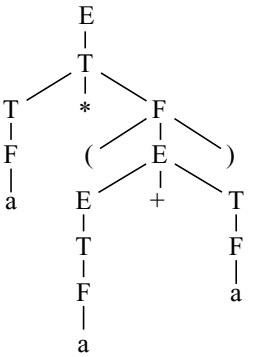


Fig. A.15 A derivation for string "$a * (a + a)$"

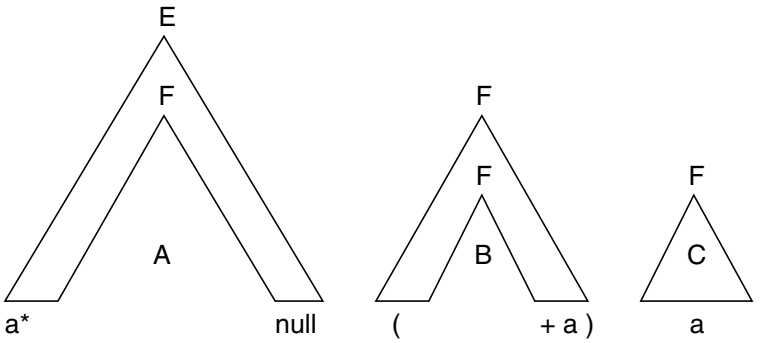


Fig. A.16 The three "jig-saw" pieces which make up string `a * (a + a)`

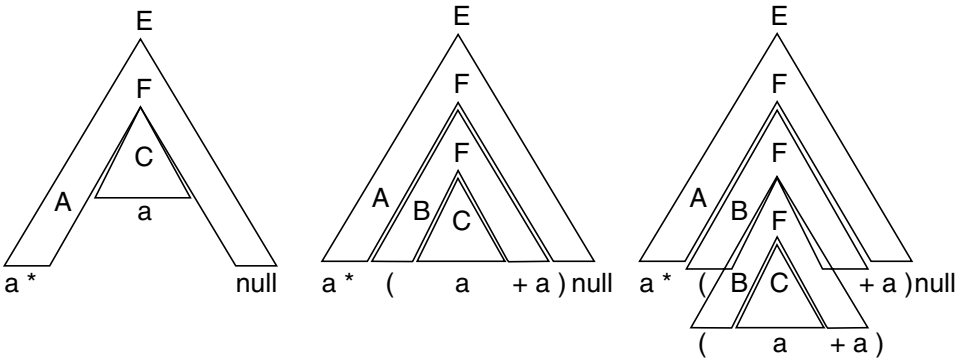


Fig. A.17 Some of the combinations of the three jig-saw pieces; first generates string `a * a`, second `a * (a + a)` and third `a * ((a + a) + a)`

This idea is the basis of the *pumping lemma* for a CFG. It will be easier if the grammar is in CNF.

Theorem A.5.5 (Pumping Lemma – 1st form) *Let $G = < V_N, \Sigma, S, P >$ be a CFG expressed in CNF, with a total of n NTs. If $u \in L(G)$, $|u| \geq 2^{n+1}$, then u may be written as: $u = vwxyz$, for some v, w, x, y and z, satisfying $|wy| > 0$, $|wxy| \leq 2^{n+1}$ and this further implies that $\forall\, m \geq 0$, $vw^mxy^mz \in L(G)$.*

See Fig. A.18, in which the shaded portion – piece *B* – can be iterated 0 or more times.

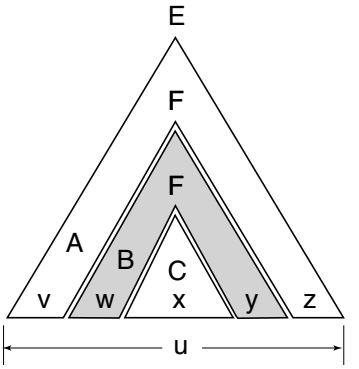


Fig. A.18 Five substrings involved in pumping lemma

Theorem A.5.6 (Pumping Lemma – 2nd form) *Here we remove the requirement that the grammar should be given as CNF. Let L be a CFL. There is an integer N such that for any $u \in L$, $|u| \geq N$, there are strings v, w, x, y and z satisfying $|wy| > 0$, $|wxy| \leq N$ and further, $\forall\, m \geq 0$, $vw^mxy^mz \in L$.*

Example

Show that the language $L = \{xayb \mid x, y \in \{a, b\}^* \text{ and } |x| = |y|\}$ is CFL by using pumping lemma.

Select $v = null$, $w = \{a, b\}$, $x = a$, $y = \{a, b\}$ and $z = b$. Then it is easy to see that the lemma is satisfied and thus *L* is CFL.

? EXERCISES

1. We delete from the set of states Q of an FSM M, all the states not reachable from q_0 and get a new FSM M_1. Is it true that $L(M_1) = L(M)$?
2. Consider a set $A = \{1, 2, 3\}$ and a relation $R : A \rightarrow A$, $R = \{(1, 2), (2, 2), (1,1)\}$. What is the nature of R?
3. In some NDFSM with ϵ transitions, there is a state with ϵ transition to itself. What is the language recognized by a new machine in which this transition is deleted?
4. State True/False:
   (a) The language $L(R)$ for any regular expression R is sometimes finite.
   (b) If R does not have any * or +, then $L(R)$ is always finite.
5. Give inductive proofs for:
   (a) If $REV(x)$ is reverse of any string x, then $REV(REV(X)) = x$.
   (b) $\forall\, n \geq 1, \sum_{i=1}^{n} 2i - 1 = n^2$.
   (c) Let a language L be defined as: (i) $a \in L$, (ii) if some $x \in L$ then $(x) \in L$. Prove that any string $x \in L$ will have at least as many '(' as there are ')'.
   (d) For $n \geq 0$ strings $A(n)$ and $B(n)$ are defined as: $A(0) = 0$, $B(0) = 1$, $\forall\, n > 0$, $A(n) = A(n-1)\, B(n-1)$, $B(n) = B(n-1)\, A(n-1)$. Prove that $A(n)$ contains neither '000' nor '111' as a substring.
   (e) Prove that the above strings $A(n)$ and $B(n)$ differ in every symbol positions.
   (f) Consider straight lines drawn in an infinite plane, such that no two lines are parallel and no three lines have a common point of intersection. The lines divide the plane in disjoint regions. Prove that the number r of such regions is given by $r = 1 + \frac{n(n+1)}{2}$. Hint: $n = 0$, $r = 1$; $n = 1$, $r = 2$.

6. Let C and D be regular expressions over some alphabet Σ. Show that the equation $R = C + RD$ is satisfied for an RE $R = CD^*$. Draw a state diagram of a skeleton FSM showing this relationship.
7. Let $\Sigma = \{0, 1\}$ be the alphabet. Develop an FSM M_1 recognizing the language $L_1 = \{x \in \Sigma^* \mid x$ begins with 00 or 11$\}$. Also develop another FSM M_2 recognizing the language $L_2 = \{x \in \Sigma^* \mid x$ ends with 00 or 11$\}$. Could M_1 and M_2 be created out of an FSM accepting the language $\{00, 11\}$? If yes, show how it can be done, if No, then prove it.
8. In some programming languages, identifiers may have embedded underscore '_' characters. However the first character may not be an underscore, nor may two underscores appear in succession. Write a regular expression that specifies such identifiers.
9. Write a regular expression that generates the Roman representation of integers from 1 to 99.
10. Write a regular expression to represent a book – with front and back covers, contents, chapters and an index. Each chapter has a section with exercises at end. There has to be at least one chapter in a book.
11. Give a recursive definition of a language $L \subseteq \{a, b\}^*$ and $\forall x \in L$, x has number of 'a' exactly double the number of 'b'.
12. Develop a PDA for accepting the language generated by the grammar:

```
E -> E - T | T
T -> T / F | F
F -> [ E ] | i
```

and demonstrate the acceptance of string $i - i/i$.

13. Develop an equivalent grammar without useless NTs for the grammar:

```
S -> ABC | BaB
A -> aA | BaC | aaa
B -> bBb | a
C -> CA  | AC
```

14. Develop a grammar in as much details as you can to represent the `fprintf(fp, "format", var-list)` function call. Note that the number of format specifiers and number of variables in var-list must match.

GLOSSARY

Table A.7 Simple meanings of some common terms

Term	Equivalent term/s	Used for
Alphabet	Vocabulary	A finite set of symbols
Argument		An input to a function
Binary relation		A relation whose domain is a set of pairs of things
Boolean operation		An operation on Boolean values

Table A.7 (Cont'd)

Boolean value		TRUE and FALSE, or 1 and 0
Cartesian product		Of sets A, B is a set consisting of all possible pairs formed by an element from each of the sets A and B
Complement		Of a set A is a set consisting of all elements not present in A, with respect to some universal set U
Concatenation		Sticking together two strings; if operating on two sets, then result is a set obtained by sticking all pairs of strings
Conjunction		Boolean AND operation
Connected graph		Has paths for each pairs of nodes
Cycle		A path through a graph which starts and ends on the same Node
Directed graph		Paths have a specified direction
Disjunction		Boolean OR operation
Domain		A set of possible *inputs* to a function
Edge	Connecting line, arc	A line in a graph
Element	Object	A member of a set
Empty set	Φ	A set having no member
Empty string	λ	A zero-length string
Equivalence relation		A binary relation which is Reflexive, Symmetric and Transitive
Function		An operation that maps inputs to outputs
Graph		A set of points and lines connecting some of the pairs of points
Intersection		Of two sets A, B is a set which have all the elements common between A and B
k-tuple		A list of k objects
Language		A set of strings
Node	Vertex	A point in a graph

Table A.7 (Cont'd)

Pair	2-tuple	A list of two objects
Path		A sequence of nodes in a graph connected by edges
Predicate		A function whose range is {TRUE, FALSE}
Range		A set of all possible values of output of a function
Relation		A predicate, whose domain is usually a set of *k*-tuples
Sequence	List	A series of objects, the order is significant
Set	Class, collection	A group of objects
Simple path		A path without repetition of any nodes
String		A finite list of symbols from some alphabet
Tree		A graph without cycles
Union		Of sets is combination of all the elements, without duplicates, into one set

APPENDIX B

Assemblers and Macro Processors

Assembly language is almost at the bottom of hierarchy of computer programming languages. It is specific to a particular family of processor, tedious to use, error-prone, but is still used for those pieces of software which requires full access to all the peculiarities of a processor. It is required to be used to implement CPU-specific functions in operating systems.

Assembly language is also used during a later step in a HLL compiler, specially to handle memory allocation function. Thus, an assembler is one of the phases of a typical compiler.

A very rudimentary version of assembly language is *mnemonic machine* language, where the op-codes of the processor are writable as mnemonics, rather than bit-patterns. The corresponding translator is called *line-by-line* assembler.

B.1 Assembly Languages

There are five major advantages of assembly language, as compared to machine language:

1. It uses mnemonic op-codes, we do not need to remember actual bit-patterns of the op-code. For example, referring to 8085 microprocessor, we can write 'ADD A' instead of '87' in hex.
2. Addresses can be given as symbolic addresses instead of numeric values. The assembler keeps track of the symbolic addresses by building a *Symbol Table*.
3. It is more human readable.
4. It makes available certain auxiliary operations like defining data and storage areas, setting the *origin* of a program segment, etc.
5. It may provide address arithmetic, macros, scoping of identifier names, conditional assembly, ready-made macros or subroutine library for input/output, etc.

There are also certain disadvantages:

1. You have to use an assembler to translate it.
2. A programmer has to mentally translate the steps of the job to be implemented into a machine-specific sequence of operations.
3. Many mundane and routine operations are to be implemented by the programmer.
4. It is tedious and error-prone. Programmer productivity is very low.
5. It is non-portable, i.e. if the CPU family is changed, complete recoding is required.

In this appendix, we shall use 8085 microprocessor machine code and assembly language as a running example. A typical small program in assembly language: this program copies 100-bytes from location 4000_{16} to 4064_{16}. The program starts at location 7000_{16}.

B.1.1 Example Assembly Language

As indicated above, we shall use Intel 8085 microprocessor's assembly language as a running example.

Statements: The statements in this language are written one per line. The structure of a statement looks like:

```
<lable> : <Mnemonic> <blank/s> <operand1>
[ , <operand2>] [; <comments>]
```

Mnemonic may be a machine code or a pseudo-operation like data definition. Labels are upto 5 characters from set [A–Z@?0–9] with first character from [A–Z@?].

Pseudo-operations: There are several pseudo-operations available:

DB – define a byte or a sequence of bytes.
DS – define storage in terms of bytes.
DW – define 16-bit word data, order is Little-Endian.
EQU – equate a symbol to some value.
ORG – set origin of the following code.

```
        ORG   4000H
BLK1:   DS    100
BLK2:   DS    100
COUNT   EQU   100
        ORG   7000H
        LXI   H,BLK1
        LXI   D,BLK2
        MVI   B,COUNT
LOOP:   MOV   A,M
        STAX  D
        INX   H
        INX   D
        DCR   B
        JNZ   LOOP
        HLT
        END
```

Fig. B.1 A small assembly language program

Note that EQU requires a label (without colons), DB, DW and DS usually have label and ORG should not have a label.

Addresses: Addresses may be expressed in any of the following forms:

Decimal: Using digits [0–9], not more than 5 digits.

Hexadecimal: Using digits[0–9A–F], not more than 4 digits, followed by 'H' and starting with [0–9].

Octal: Using digits [0–7], followed by 'O' or 'Q'.

Binary: Using digits [0–1], followed by 'B'.

Symbolic: A data label defined somewhere in the program. Address arithmetic may be done – allowed operations are +, −, MOD, *. All address calculations using 16-bit numbers.

Offset from Location Counter (LC): LC is denoted by '$' in the operand field. Offset can be positive or negative.

B.2 Assemblers

Jobs to be done by an assembler: An assembler has to do the following at the minimum:

1. Scan the input assembly language code, check that only valid characterset is used and statement constructs are valid.
2. Replace all mnemonic op-codes by correct machine op-codes.

3. Replace all symbolic addresses, address constants, etc. by proper numeric values.
4. Assign data areas and provide for loading-defined data wherever specified.
5. Execute all the other pseudo-operations.

We shall now consider various types of assemblers and their major characteristics. We then discuss working of a two-pass assembler for the example assembly language.

Types of Assemblers

- **Line-by-line assembler:** This type of assembler is usually used in debuggers, where the test engineers may like to place short code segments in a program being debugged. Only mnemonic op-codes are provided. Address are all numeric. No symbol table is created or looked up. We shall not discuss this type of assembler further.
- **One-pass assembler:** The assembler scans through the assembly source code only once and is expected to assemble the program during that single pass.
- **Two-pass assembler:** The assembler scans through the complete source code two times. During the first pass, a *Symbol Table* is builtup and during the second pass actual assembly of each instruction takes place, using the Symbol Table to provide address equivalents for the symbols.
- **Macro assembler:** An assembler which handles macros also. We discuss macros and macro assembler in Section B.2.4.
- **Cross-assembler:** The assembler which run on a machine other than the one for which it assemble the object program (see Fig. B.2).

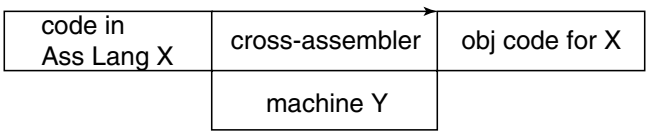


Fig. B.2 T-diagram of a cross-assembler

B.2.1 One-pass Assembler

Such an assembler basically cannot handle forward references. For example, consider the code:

```
DCR        B
            JZ    FRWRD
            ...   ...
            ...   ...
FRWRD:      ...   ...
```

Here, the one-pass assembler will not have address for label FRWRD and so will not be able to assemble the JZ instruction. There are three options for such an assembler:

1. Forbid the programmer from using a forward reference. This of course cripples the language and is not an acceptable option.
2. While reading the source code, go on storing the statements on a secondary storage unit like disc or tape. During a later phase during which the forward references can be resolved, read the source from this saved copy. This option, actually is a two-pass assembler, is disguise and is irrelevant in modern days, where fast hard discs are almost a standard feature of a general-purpose computer.

3. If a forward reference is found, add a node to a 'ToDereference' array of linked list, where each array element has a structure:

   ```
   <symbol> <address> <ptr> --> <obj code pos #1> <obj code pos #2> ...
   ```

   When a forward reference is not yet defined, <address> is zero. When it gets defined later in the code, the assembler inserts its address value at the code positions according to the link list. This assumes that either the complete object code is held in memory or a "random-load" loader is required.

Such an one-pass assembler was relevant when in old days the secondary storage units were slow and expensive. Sometimes it is used as back-end of a compiler, which maintains its own symbol table, though the tendency is to use a "standard" two-pass assembler even there.

We shall not discuss one-pass assembler any further.

B.2.2 Two-pass Assembler

The assembler has two passes or scans over the input source code. During the first pass, the emphasis is on building up a symbol table apart from checking the lexical validity of the code. During the second pass, it replaces all references with their equivalent addresses or values, assembles the code and outputs it.

The assembler uses the following data structures:

1. **Symbol Table:** The structure is:

   ```
   <symbol> <value>
   ```

2. **Op-code table:** Stores the base values only for each op-code; the actual op-code may be computed using the modifiers from register value table. The structure is:

   ```
   <Mnemonic> <Hex value> <no. of oprnds> <Type op1> <Type op2> <length>
   ```

3. **Register value table:** (operand modifier in general) The structure is:

   ```
   <Reg name> <1st value> <2nd value>
   ```

4. **Error Table:** Error codes and corresponding legends.

The algorithms for Pass-I and Pass-II are given as Algorithms B.2.1 and B.2.2, respectively.

Algorithm B.2.1: Pass-I

```
LC ← 0;
address ← 0;
scan next line of source;
while not END do
   if label then
      if EQU then
         enter label in SymTab with oprnd value;
      else
         enter label in SymTab with LC value;
      end
   else
```

Algorithm B.2.1: Pass-I (*Cont'd*)

```
        if ORG then
            LC ← origin vale;
        end
    end
    increment LC as per length;
    scan next line of source;
end
goto Pass-II;
```

Algorithm B.2.2: Pass-II

```
LC ← 0;
scan next line of source;
while not END do
    if ORG then
        LC ← value;
    else
        if DS then
            find v = oprnd value;
            length ← v;
        else
            if DB then
                find oprnd values and length;
            else
                if DW then
                    find oprnd values and length;
                else
                    get op-code;
                    if valid then
                        if oprnd reqd then
                            if symbolic, get from SymTab;
                            assemble code and output;
                        else
                            store binary at LC;
                        end
                    else
                        print ERROR;
                    end
                end
            end
        end
        increment LC by length;
    end
    scan next line of source;
end
```

B.2.3 Symbol Table Handling

One of the most frequent operations and one which consumes major portion of execution time in an assembly operation is Symbol Table building and look-up. A real-life assembly language program may have hundreds or thousands of symbols in use. The data structure used to store the table and the algorithms used to access it contribute significantly to the speed of an assembler. The operations done on a symbol table are:

1. insert a symbol if not already present in the table;
2. look up a symbol if it is present in the table;
3. look up various attributes of a symbol;
4. modify attributes of a symbol;

Note that deleting a symbol is not a required operation. The data structure should be selected keeping these points in mind.

What are the possible options for the data structure? They are:

Simple array: Each entry is a (symbol, value) pair. As the array size would be limited and fixed, useful only for experimental work, because the number of symbols that can be allowed in the source will be limited by the array size. Also, if the symbol names are stored directly in the array, there will be an upper limit on the size of the names also, considerable wastage of memory.

Array with pointers to symbol names: The symbol names are stored separately in a character array and within the main symbol table we put the starting index and length of the symbol name. This will allow considerable variation in the size of the symbol names, at the same time saving memory (see Fig. B.3). Even with this enhancement, the major problem of limited number of entries remains. If the implementation language is C, we can use '0' terminated C strings in the name storage. This avoids storing of length value in the symbol table entry, but the implementor must take care to use the character array as readonly once a symbol name is entered in it, otherwise there is a danger of disturbing the array.

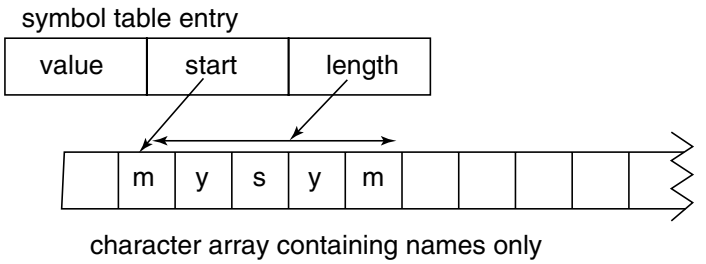


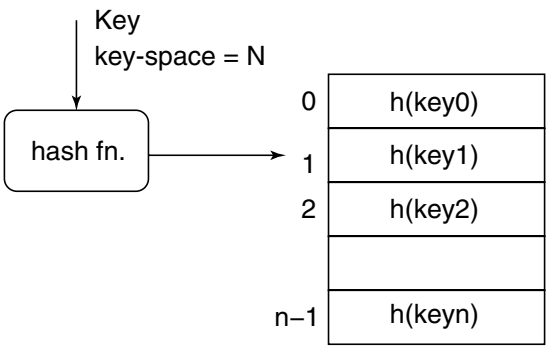
Fig. B.3 A symbol table entry using separate name storage

Sorted array: If the entries are put in the symbol table in an arbitrary order, then for each look-up, time $O(n)$ will be taken, where n is the number of symbols in the table. We can sort the array every time a new symbol is inserted and use the *Binary search* method to look-up the table. As there is only one time insertion for each symbol, but there may be several look-ups for it, this seems to be a viable approach. As sorting does take time, one alternative is to sort the table only after 10 new symbols are inserted and use an auxiliary linked list to keep new symbols. Look-up operation checks the linked list first and then the main sorted array. When linked list has 10 symbols in it, all of them are put in the array and the array is resorted.

Linked list: If the symbol table is constructed as a singly linked list, one of the major objections to the array storage does not apply. The loop-up times will be, though, large and unacceptable for large symbol tables. One way out of this is to use multiple linked lists, one each for symbol name starting with a particular character. The link head pointers are kept in an array.

Binary search tree: Provide considerable improvement, at the cost of complexity of handling routines, over the above methods. Here, assumption is that the new symbols arrive in a random lexical order, so that the BST is more or less balanced.

Hash table: Provide possibly the best performance, especially for large source codes having thousands of symbols. Here, the main idea is that each symbol name is converted to a corresponding *hash value* by a fast *hashing* function. The hash value is a positive integer $\leq$ the size of the hash table. The symbol data is inserted in the hash table at that index position (see Fig. B.4).



N >> n, unique mapping not possible

Fig. B.4 Hash table: basic idea

Unfortunately, the symbol name space will generally be very large compared to the number of entry slots available in the hash table. This results in what is known as *collision*. One of the popular methods to handle these inevitable collisions is to use linked list for each of the hash value, as shown in Fig. B.5.

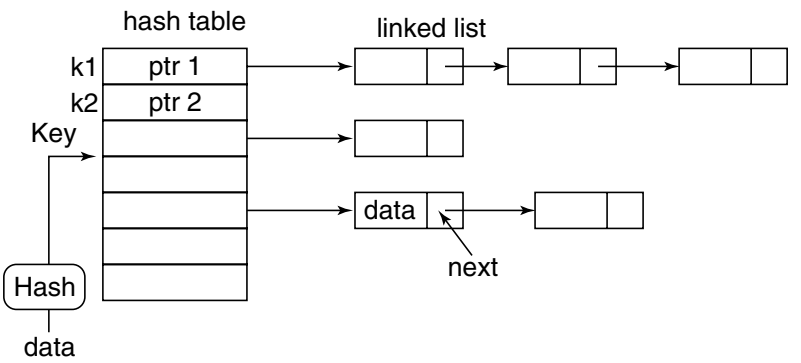


Fig. B.5 Hash table: collision resolution

B.2.4 Macro Assembler

It is usual to have a very useful facility, known as *Macro* facility, in an assembler. Though macro facility can be made quite complex, and many real-life assembler do have sophisticated macro facilities, even simple macro implementation can be of considerable help to an assembly language programmer.

What is a Macro?

A Macro simply means a name given to some pattern. It refers to a mapping *macro : symbol* ⇒ *pattern*. We say that the pattern *defines* the macro denoted by the symbol. There are two steps in using a macro: first, define a macro, by associating a symbol with an arbitrary pattern, then second, use the macro, by simply using the symbol. When we use the macro, *macro expansion* or replacement takes place, i.e. wherever the symbol is occurring, it is replaced by the pattern.

This simple idea of replacement of a symbol by a pattern is very powerful. It is used in assemblers, compilers, spreadsheets, text editors, etc. In fact, the software that the authors of this book used to compose it – TEX and LATEX uses macros very heavily. In Section B.3, we discuss several such macro processors.

Macros in an Assembler

An assembly language programmer finds that he uses certain sequences of assembly language statements repeatedly, with possibly minor variations.

For example, assume that adding 4 to the value of H register occurs several times in our program. We may define a macro as follows:

```
ADD4   MACRO
       MOV A,H
       ADI 4
       MOV H,A
       ENDM
```

Here, ADD4 is the name of the macro, i.e. the symbol to be replaced during macro expansion, MACRO and ENDM are macro assembler pseudo-operations denoting beginning of macro definition and end of definition respectively.

If this macro definition is included in an earlier part of our program, then later on we can use it simply as:

```
...    ...
ADD4
...    ...
```

and it will result in the following code, ADD4 being replaced by 3 lines of code.

```
...    ...
       MOV A,H
       ADI 4
       MOV H,A
...    ...
```

Note well that pseudo-operations MACRO and ENDM are not inserted. This insertion happens every time the symbol ADD4 is used.

Let us try to define another macro – Load accumulator indirect, which is a very common operation. We would like to load data from a specified address into the *A* register. This requires that we use a macro argument to be able to specify any address.

```
LDIND   MACRO  ADDRS
        LHLD   ADDRS
        MOV    A,M
        ENDM
```

The symbol ADDRS is the local macro argument. While using the macro, it will be replaced by a use, call or invocation argument. Thus,

```
...      ...
LDIND    1234H
...      ...
```

Note that we have supplied the call argument as 1234H. The code inserted will be:

```
...      ...
LHLD     1234H
MOV      A,M
...      ...
```

As a further example, the bytes copy operation as shown in Fig. B.1 can be implemented as a macro:

```
MOVBT   MACRO    BLK1,BLK2,COUNT
        LXI      H,BLK1
        LXI      D,BLK2
        MVI      B,COUNT
LOOP:   MOV      A,M
        STAX     D
        INX      H
        INX      D
        DCR      B
        JNZ      LOOP
        ENDM
```

Note the three macro arguments BLK1, BLK2 and COUNT. A difficulty arises if we try to use this macro in our program at more than one place. The label LOOP will be defined multiple times. The first thing to note is that the macro arguments BLK1, BLK2 and COUNT are *local* to the macro and they will be replaced by the call arguments, so there is no problem with them. Can we tell the macro processing functions within the macro assembler to create unique labels *on fly?* Yes, if we write $<number> instead of LOOP as the label, the macro assembler will generate unique labels for each of them every time the macro is used and insert them in the symbol table.

```
     ...     ...
     MVI     B,COUNT
$1:  MOV     A,M
     STAX    D
     ...     ...
     JNZ     $1
     ENDM
```

A macro assembler may allow calling a previously defined macro within the definition of another macro (*nested macro call*). It may further allow definition of a macro within definition of another macro (*nested macro definition*.)

Macro Processing

We shall assume that the macro processing is done independently of the assembler, as a pre-processing step. Also, we discuss only a simplified version, without nested macro calls or nested macro definitions. We have two passes – Pass-I will search for macro definitions and store them; Pass-II detect macro calls and expand them. The data structures used are:

MDT Macro Definition Table – stores the body of macro definitions.

MNT Macro Name Table – stores names of defined macros.

MDTC MDT counter – indicates the next available entry in MDT.

MNTC MNT counter – indicates the next available entry in MNT.

ALA Argument List Array – stores macro arguments and call arguments.

MDTP MDT pointer – indicates the next line of text during expansion.

The algorithms are given as Algorithms B.2.3 and B.2.4.

Algorithm B.2.3: Macro processing Pass-I

```
MDTC ← 1;
MNTC ← 1;
read next source line;
while not END do
   if MACRO then
      read next source line;
      MNT(MNTC) ← name,MDTC;
      increment MNTC;
      prepare ALA;
      MDT(MDTC) ← name;
      increment MDTC;
      read next source line;
      while not ENDM do
         substitute indices for arguments;
         enter line into MDT;
         increment MDTC;
         read next source line;
      end
   else
      write copy to intermediate file;
   end
   read next source line;
end
goto Pass-II;
```

Algorithm B.2.4: Macro processing Pass-II

```
read next source line;
while not END do
    search MNT for match;
    if name found then
        MDTP ← MDT index from MNT entry;
        set up ALA;
        increment MDTP;
        get line from MDT;
        while not ENDM do
            substitute arguments from macro call;
            write to expanded file;
            increment MDTP;
            get line from MDT;
        end
    else
        write to expanded file;
    end
    read next source line;
end
send expanded file to assembler;
```

Macros and Subroutines

Seemingly, from the viewpoint of the programmer, macros provide the same kind of facilities as subroutines. Then how do they compare?

Macro: Complete macro body code gets substituted every time it is used, so increases the code length. As it is *in-line* code, save on the subroutine call and return instruction time and stack load overheads, and thus works fast.

Subroutine: Code is included only once in the program. Time and stack-space overheads for each call and return.

In summary, we can say that if the job can be done with a few instructions, say 5 to 10, then it may be advisable to implement it as a macro, otherwise it should be implemented as a subroutine.

B.3 Macro Processors

In this section, we have a brief look at some real-life macro processors. The first one is a general-purpose macro handling facility, while the second one is for a specific language use.

B.3.1 Real-world: M4 Macro Processor

This macro processor is a general-purpose macro processor, used extensively on Unix and Unix-like systems, including Linux. An extended version is available as a GNU package.

M4 is a macro processor, in the sense that it copies its input to the output, expanding macros as it goes. Macros are either built-in or user-defined, and can take any number of arguments. Besides just doing macro expansion, M4 has built-in functions for including named files, running shell commands, doing integer arithmetic, manipulating text in various ways, performing recursion, etc. It can be used either as a front-end to a compiler or as a macro processor in its own right.

It is widely available on all UNIXes and has been standardized by POSIX. It is used by GNU Autoconf.

A very simple example M4 can be used for simple embedded text replacement. If M4 receives the input

define(AUTHOR, William Shakespeare) A Midsummer Night's Dream by AUTHOR

then it outputs

A Midsummer Night's Dream by William Shakespeare

B.3.2 Real-world: Pre-processing (cpp)

The C pre-processor, often known as cpp, is a *macro processor* that is used automatically by the C compiler to transform the program before compilation.

It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs. These macros are expanded in its output. It also allows "inclusion" of other source files. Additionally, it provides *conditional compilation*. It also removes most of the comments.

Some examples

```
#define NDEBUG
#define SIZE 100
#define ARRAY SIZE*4
#define max(x,y) ((x)>(y))?(x):(y)
#include <stdio.h>
#include "myheader.h"
#ifdef LINUX
#endif
```

defines (macro)

```
#define BUFFER_SIZE 1024
```

later in the source:

```
buffer = (char *) malloc (BUFFER_SIZE);
```

Replaced by pre-processor:

```
buffer = (char *) malloc (1024);
#define NDEBUG
```

Only NDEBUG is "defined" but it has null (no) value.

```
#define BUFFSIZE 4*SIZE
#define SIZE 10
```

what happens? is it valid?

Defines with Arguments

```
#define power2(x) (1<<(x))
```

when invoked later in the source code with

```
a = power2(3);
```

is replaced by pre-processor:

```
a = (1<<(3));
```

and compiler replaces by $a = 8$; It is better to put the argument in () in the expansion (why?).

More than one formal argument can be used. The (in the define should immediately follow the macro name

```
#define power2 (x) (1<<(x))       what happens?
```

Define ends with new-line

The expansion part of a "define" ends on the new-line character. How to circumvent this?

```
#define NUMBERS  1, \\
                 2, \\
                 3
```

Later in the source code, we have

```
int x[] = { NUMBERS };
```

It is replaced by:

```
int x[] = { 1, 2, 3 };
```

include Directive

```
#include <FILE>
```

This variant is used for system header files. It searches for a file named FILE in a standard list of system directories. You can prepend directories to this list with the `-I` option.

```
#include "FILE"
```

This variant is used for header files of your own program. It searches for a file named `FILE` first in the directory containing the current file, then in the same directories used for `FILE`.

Inclusion can be nested, to any reasonable depth.

The Standard list for UNIX and UNIX-like systems

```
/usr/local/include
/usr/lib/gcc-lib/TARGET/VERSION/include
/usr/TARGET/include
/usr/include
```

`TARGET` refers to the processor and platform, e.g. on a typical machine it is i486-slackware-linux
`VERSION` is the gcc version, e.g. 3.2.3.

Conditional Directive

A directive that instructs the pre-processor to select if a portion of code is included in the final token stream passed to the compiler. It can test arithmetic expressions, or whether a name is defined as a macro, or a mix using the special "defined" operator. Resembles in some ways an if statement in C, but: an if statement is tested during the execution of your program, a pre-processor conditional directive is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Conditionals are:

```
#if         test any (restricted) integer expression
#ifdef      test if a macro is defined
#ifndef     test if a macro is not defined
#endif      end of the #if group
#else       if the condition being tested is not true
#elif       if condition not true, open a new if group
defined     used in #if to check if a macro defined
```

Other Facilities in gcc Pre-processor

There are a number of other facilities in the pre-processor (cpp) of gcc, like:

Stringification – convert expressions to strings

Diagnostics – provide pre-processor level messages

Line control – provide line number info to other phases

Pragma – define compiler-specific information etc.

Please refer to cpp documentation by looking up the man-pages.

? EXERCISES

1. Find out how can you know the set of ALL files are included during a compilation.
2. There are some pre-defined "defines". How can you find out which are they?
3. What happens if file `myA.h` contains

   ```
   #include "myB.h"
   ```

   and `myB.h` contains

   ```
   #include "myA.h"?
   ```

APPENDIX C
Linkers and Loaders

C.1 Linkers

In any real-life application development, the total software is divided into a number of modules, which are developed, debugged and tested separately and then joined together. Each module may be made up of several sub-modules, etc. All these modules need to be *linked up* to form a single executable program. At the minimum, even for a student's single module learning program, at least the run-time library need to be linked with it.

Linkers are programs that link together several object modules and libraries to form a single, coherent program. Object modules are the machine code output from an assembler or a compiler and contain executable machine code and data together with information that allows the linker to combine the modules together to form a program. For example, one module might contain all of a program's database functions and another module its command line argument handling functions. Linkers fix up references between these object modules, where a routine or data structure referenced in one module actually exists in another module.

C.1.1 Relocation and Linking

When a program is to be loaded in the memory in preparation for its execution, several issues arise:

- A program or data module, assumed to be beginning at a particular memory address *S* in the source code, is ultimately to be loaded in the memory at some different position *L*. This will require *relocating* the modules.
- The program may be made up of several separate object modules. They will require joining or *linking*.
- The program will need one or more *libraries* to provide functions assumed in the source language, but not available in the hardware or the operating system. These libraries need to be made available by effectively linking with modules, this is linking operation. In case of multi-tasking operating systems, it is desirable that only a single copy of the libraries be kept in the memory and all running programs *share* them. This requires that the libraries are *re-entrant* and position independent.
- There may be references to identifiers defined in some other modules from a module.
- There may be some identifiers defined in this module, which may be accessed from some other modules.

Relocation

For modern programming languages, we require a relocating and linking loader. Compiler should generate Relocatable and Linkable object code in a specific format.

Relocatable means that the object code should be in such a form that it can be executable after either:

- loading anywhere in the memory (position independent code);
- loading and modifying some of the addresses.

See Section C.4 on "Loaders".

Linking

Linkable means that references to data and program entry points in one module be possible from other modules. The process of establishing the means of such references in separately compiled modules is called linking. Thus, the object code should have the necessary information by which linking is possible (see Fig. C.1).

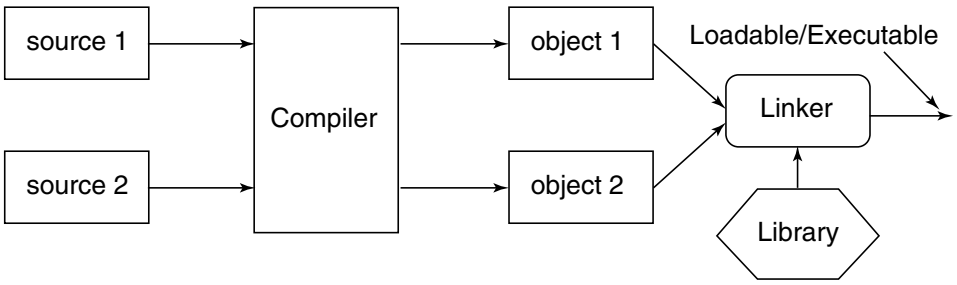


Fig. C.1 Linking process

Example: As an example, consider some modules to be linked, see Fig. C.2.

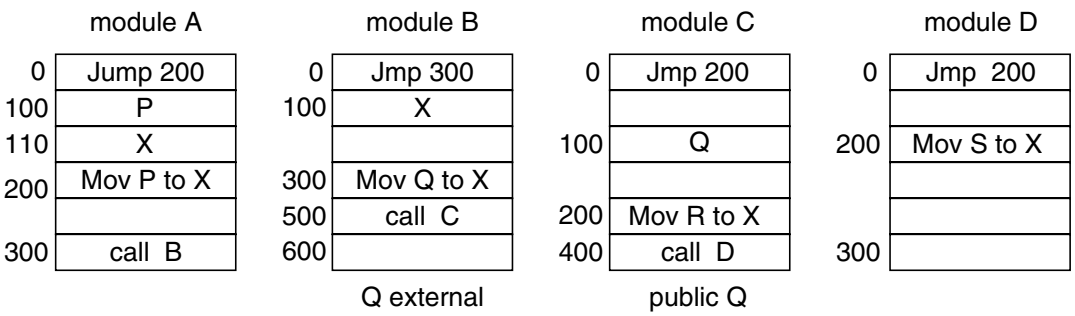


Fig. C.2 Several modules to be linked

There may be jumps within modules, like in module *A*. There may be reference to external symbols, like in module *C*. There may be local reference to symbols, like in module *A*. There may be call to external subroutines, as in modules *A*, *B* and *C*.

C.1.2 Tasks Performed by a Linker

Each module starting address is 0, each is to be allocated a separate load address.

For the example modules in Fig. C.2, assuming that the O/S resident part takes 100 units, then the load addresses are as shown in Table C.1.

Obviously, the Linker will have to:

- adjust the Jump addresses;
- adjust address of local references;
- find location of the external symbols and insert their addresses.

Table C.1 Example: modules to be linked

Module	Length	Starting or Load address
A	300	100
B	600	400
C	400	1000
D	300	1400

C.1.3 An Object Module

What information should be there? See Fig. C.3.

Contents of each section of an object module:

Header – lengths of each of the remaining sections and length of the uninitialized data area (bss);
Machine code – generated binary executable code; `(txt)`
Initialized data – data values known to the compiler, constants and initialized variables; `(data)`
Symbol table – local symbols, undefined external symbols, public symbols.
Relocation information – including linking information:

1. A list of all the symbols and their positions within this module which can be referenced by other modules. `(esd)`
2. A list of all the symbols not defined in this module but referenced. `(esd)`
3. A list of address constants, their positions and how to adjust them. `(rld)`

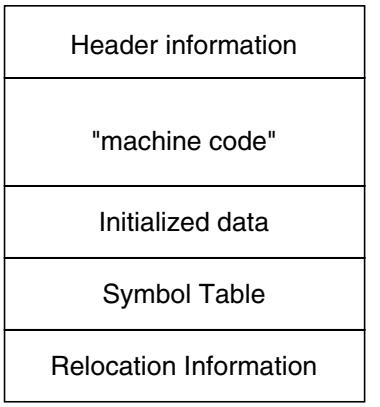


Fig. C.3 Minimum information which needed to be present in an object module

Note that uninitialized data – declared variable areas – do not appear in the object module, only its size is needed. This is so because really speaking when this area is defined in an actual memory at program load time, it cannot be uninitialized: some data is always there in memory (may be garbage, may be not!)

In the above list, `(esd)` denotes External Symbol Dictionary, containing those symbols which may be communicating with external modules. The Relocation and Linkage Directory `(rld)` contain information about those locations within the program which are dependent on the load address. There may be one or more `esd` records and one or more `rld` records in an object module.

C.1.4 Logical View of Process Memory

Roughly speaking, a *process* is a running program. When a process is active, its memory layout is shown in Fig. C.4. Compare the regions with those given in Fig. C.3.

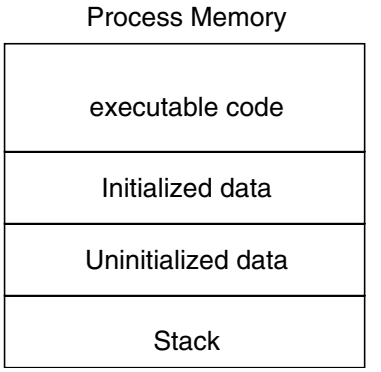


Fig. C.4 Logical view of process memory

C.1.5 Actual Process Memory

Actual process memory has one more component – free memory or *heap area*. This is the memory area from which portions are allocated when library function calls like `malloc()` are executed. Because these memory area allocation requests are received by the system not at compile time, but at execution time, the compiler cannot pre-assign the memory as in the case of initialized or uninitialized data areas.

There are two kinds of linking possible in modern Virtual Memory based systems: Static linking and Dynamic Linking. Before we discuss working of a linking loader, we need to understand these two types of linking.

C.1.6 Static Linking

In this case, all the modules, along with the needed libraries, are linked together as the last step of compilation (see Fig. C.6).

Static linking results in a much larger sized executables as all the stuff needed to execute the program is in one place, the executable file. This increases the program load time also. On the other hand, the executable is more portable, as it depends upon certain system calls only and not on particular versions of the libraries.

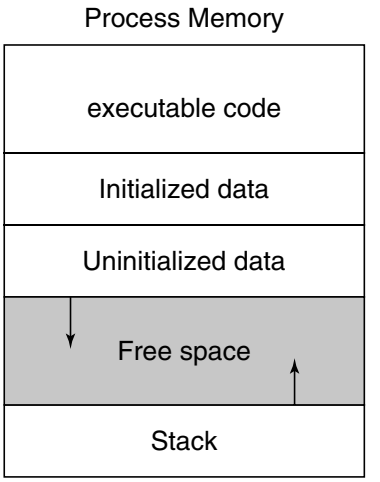


Fig. C.5 Actual process memory. Note the "free" area

C.1.7 Dynamic Linking

In this case, the user's object module is "linked" to the entry points and external symbols in libraries, in the sense that required address adjustments are done, but the actual code is not joined with the object module. Thus, the resultant "executable" is much smaller, but needs to be joined at program load time with the libraries (see Fig. C.7).

This also results in the executable being less portable across library versions. Normally, such an executable is loaded faster, as most of the libraries are anyway present somewhere in the memory, serving some other processes.

C.2 A Typical Linking Loader

We shall now describe in brief a typical linking loader.

C.2.1 Data from Object Modules

The information concerning linking operations is provided by the `esd` and `rld` records from object modules.

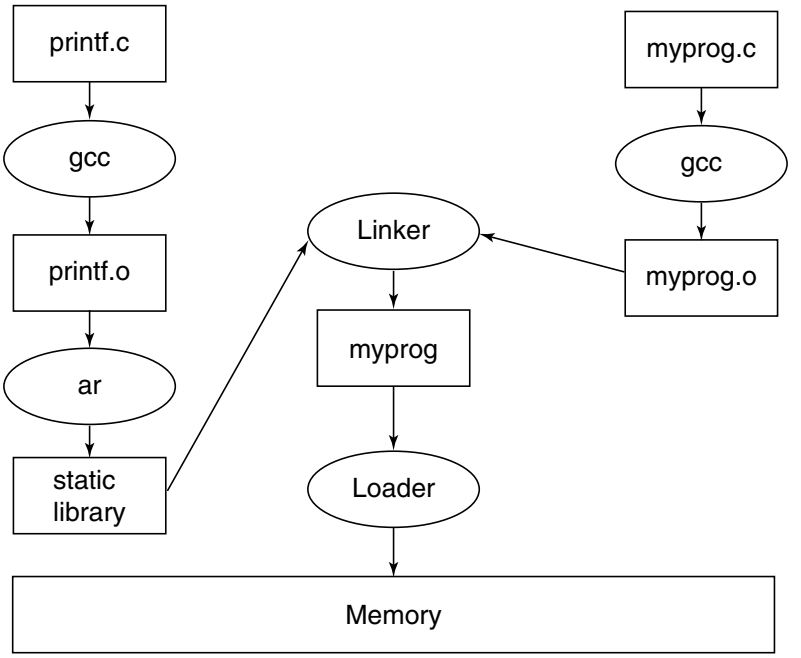


Fig. C.6 Static linking

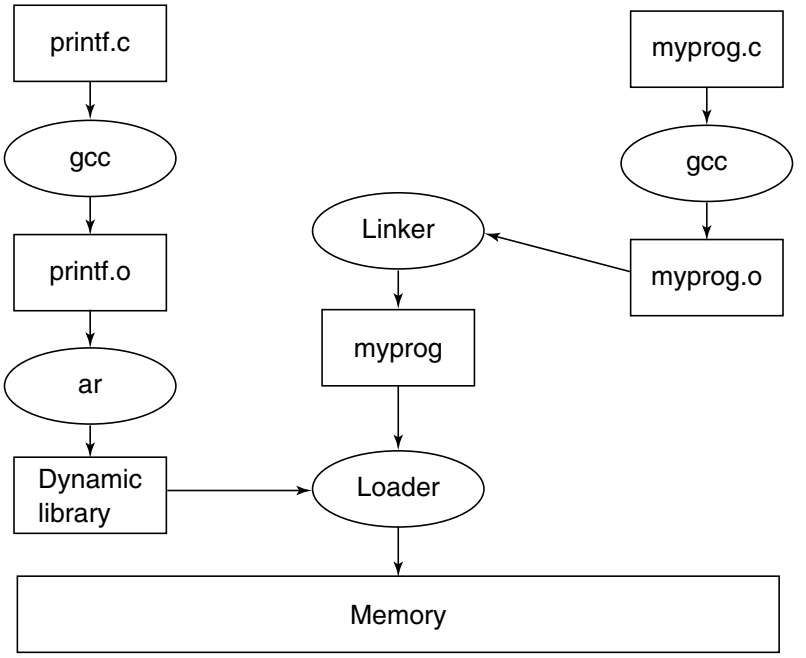


Fig. C.7 Dynamic linking

ESD

Provide the information necessary to build the External Symbols Dictionary or the *symbol table*. There are three types of external symbols:

Module Definition, MD Name of the module;
Local Definition, LD Label of a locally defined area;
External Reference, ER A symbol which is an MD or LD in exactly one place in some other module.

Each MD and ER is identified by an integer (ID) used for reference in an `rld` record.

RLD

Records have the following information:

1. The location and size of each address constant that will change due to relocation or linking.
2. The external symbol by which the address constant should be modified.
3. Whether plus or minus operation is to be done.

A linker operates in two phases – Pass-I and Pass-II.

C.2.2 Database Used or Created During Linking

IPLA Initial program load address – supplied by either the O/S or the programmer.
PLA Program load address – keeps track of each module's assigned location.
GEST Global external symbol table – stores each external symbol and its assigned address. This is prepared by Pass-I.
Load-map A listing showing each external symbol and its assigned value.
LESA Local external symbol array – used to establish a map between the ESD ID numbers and corresponding external symbol's address value.

C.2.3 Linker Algorithms

Algorithm C.2.1: Linker Pass-I

```
PLA ← IPLA;
while not EOF do
    read next record;
    if module end then
        PLA ← PLA + module-length;
    end
    if ESD record then
        if MD then
            value ← PLA;
            module-length ← length;
        end
        if LD then
            value ← PLA + addrs;
        end
        if ER then
            continue;
        end
```

```
      if symbol in GEST? then
         print "ERROR: duplicate name";
      else
         put symbol in GEST;
         print symbol, value;
      end
   end
end
goto Pass-II;
```

Algorithm C.2.2: Linker Pass-II

```
PLA ← IPLA;
extaddr ← IPLA;
while not EOF do
   read next record;
   if ESD record? then
      if MD? then
         module-length ← length;
         LESA(ID) ← PLA;
      end
      if ER? then
         search symbol in GEST;
         if found? then
            LESA(ID) ← value;
         else
            print "ERROR:undefined external";
         end
      end
   end
   if TXT record? then
      copy BC bytes from record to (PLA+addrs);
   end
   if RLD record? then
      value ← LESA(ID);
      flag = +:(PLA+addr) ← (PLA+addr) + value;
      flag = –:(PLA+addr) ← (PLA+addr) – value;
   end
   if module END? then
      if addr non-blank? then
         exaddr ← (PLA+addr);
      end
      PLA ← PLA + module-length;
   end
end
jumpto exaddr;
```

C.3 In Linux World

The Linux operating system usually uses GNU binary utilities like GNU C compiler `gcc`, linking loader `ld`, library archiver `ar`, etc. The linker-loader is quite sophisticated and has many options to control its actions.

Each process is allocated upto 4 GB of virtual memory address space, out of which the highest 1 GB address space, where the kernel routines execute on behalf of the user process, is invisible to the users program. This part of the address space is common between all the processes. The following are the general properties of Linux environment:

- loader does not necessarily load full content to physical memory, but allocates virtual memory pages;
- uses ELF (Extensible Loader Format);
- kernel virtual memory is invisible to user-mode code;
- stack at top of user-mode virtual memory, grows down;
- program text is kept in write-protected regions;
- run-time memory (heap) grows upwards.

See Fig. C.8.

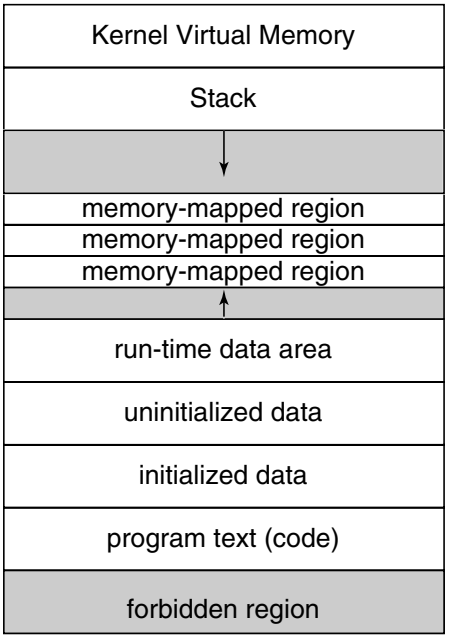


Fig. C.8 Memory layout for ELF

C.3.1 `ld` – GNU Linking Loader

The linking loader used in Linux is `ld`. The executable file format used is ELF. Several programs which help in program development are as follows:

nm – List symbols from object files.
ar – Create, modify and extract from archives.
readelf – Displays information about ELF files.
ldd – Display shared library dependencies.
objdump – Display information from object files.
ranlib – Generate and insert an index in an archive file.

C.4 Loaders

A loader is a utility program which reads an executable program file in a particular format from a secondary storage unit, puts proper bit-patterns in the memory at specified place and then initiates execution of that program. It should perform the following:

- Request *allocation* of memory space.
- Resolve symbolic references between object files. We have already seen this function, called *linking*, under "Linker", Section C.1 and is included in the loader if it is a *Direct-linking* loader.

- Adjust all load address-dependent locations – address constants, pointers, even address operands in instructions – to correspond to the allocated load address. This is called *relocation*. Most of the modern general-purpose computers use Virtual Memory organization, which essentially eliminates these requirements. We have already seen this function under "Linker" in Section C.1.
- Physically, place the program instructions and data, which is in the format shown in Fig. C.3 on the secondary storage unit, into the main memory and create a load-map similar to Fig. C.5. Again, if the system is using Virtual Memory, then the loader prepares a "blueprint" (i.e. layout) of program memory allocation, but at the start of the program, only a small portion of the physical memory is actually made available to the program, see "Virtual Memory and Loader" in Section C.4.3.

C.4.1 Absolute Loader

This is the simplest of loading scheme. The loader simply reads the executable program file and puts the program "text" (instructions) and data at the specified locations. A typical absolute loader is for `Intel Hex format` files. The input file consists of a number of records, whose format is:

```
: BC AAAA TT DD DD ... ... CH
```

where BC – two hex characters giving byte-count of this record;
AAAA – four hex characters give the load address;
TT – two characters give record type – 00 means data record, 01 means end of file record;
DD – two hex characters per data byte, maximum 16-bytes;
CH – two hex characters check-sum.

The absolute loader program is quite simple, see the pseudo-code in Fig. C.4.1.

Algorithm C.4.1: Absolute loader

```
Initialize;
while not last record do
    read next record from file;
    get load address;
    get byte-count;
    read data bytes;
    check the check-sum;
    put in memory;
end
get start address from last record;
jump to that address;
```

Advantages: Simple in concept and simple to implement.

Disadvantages: The programmer is responsible for assigning all memory addresses, useful only on very simple computer systems.

C.4.2 Relocating Loader

Here, the load addresses given in the program file are relative addresses only, generally with respect to the program beginning address of 0. The operating system will allocate a load address to the loader and the loader will load the program instructions and data at places in memory with this load address as beginning. In order that the program is still valid at the actual load address, all position-dependent addresses must be modified by the loader. For example, consider a small program segment:

```
       in program file                      in memory
0000   JMP      0020                 1000   JMP       1020
0010   0100                          1010   1100
0014       Y                         1014      Y
0020   LD       0014                 1020   LD        1014
0024   ADD  I   0010                 1024   ADD I     1010
       ...      ...                         ...       ...
0100       X                         1100      X
```

The system has assigned 1000 as the load address for the program. The relocating loader will have to modify:

1. address operand in JMP instruction, to actual program start address;
2. address constant – address of variable *X* – is to be modified;
3. address operand in LD instruction is to be modified;
4. address operand in ADD indirect instruction is to be modified.

If the CPU of the computer system supports Position Independent Code (PIC), a major portion of these adjustments can be avoided and thus reduces the program load time. PIC is generally available as one of the addressing modes. The address operands are displacements with respect to the Instruction Pointer (Program Counter). Assuming PIC address mode in effect for all the instructions, the program will look like:

```
       in program file                      in memory
0000   JMP      0020                 1000   JMP       0020
0010   0100                          1010   1100
0014       Y                         1014     Y
0020   LD       -6                   1020   LD        -6
0024   ADD  I   -14                  1024   ADD I     -14
       ...      ...                         ...       ...
0100       X                         1100     X
```

Note that even with PIC, the address constant at 0010 required to be modified by the loader, as it is address-dependent data and not an instruction. The address operands within instruction will generate proper effective memory addresses, for any load address due to the PIC mode.

The problem of adjustment of address constants is resolved if the CPU has *two-component* addressing mode available. In this mode, an effective address (EA), i.e. the operand address sent to the memory, is made up of content of a *base* register and a displacement value:

EA = (BASE) + Displacement. When a program is loaded in the memory, the relevant base register is also loaded with load address. For example, the program shown above will now look like:

```
        in program file              in memory
0000    JMP     0020         1000    JMP     0020
0010    0100                 1010    0100
0014         Y               1014     Y
0020    LD      -6           1020    LD      -6
0024    ADD B I   -14        1024    ADD B I -14
        ...     ...                  ...     ...
0100         X               1100    X
```

The 'B' flag in the ADD instruction indicates that the address constant at displacement of −14 with respect to IP is taken to be displacement with respect to the Base register. Thus, EA = (IP − 14) + (BASE) which gives the correct value 1100.

C.4.3 Virtual Memory and Loader

With infinite memory, many of the things that an loader has to do would be redundant. One of the basic tricks of any operating system is the ability to make a small amount of physical memory behave like rather large memory. This apparently large memory is known as *virtual memory*. The idea is that various programs running in the system (i.e. processes) are fooled into believing that they are running in a large memory, typically a few Gigabytes.

The system divides the virtual memory into easily handled pages and swaps these pages to and from a hard disk partition as the system runs. A process does not notice because this swapping is done when the process is waiting its turn for use of the CPU (see Fig. C.9).

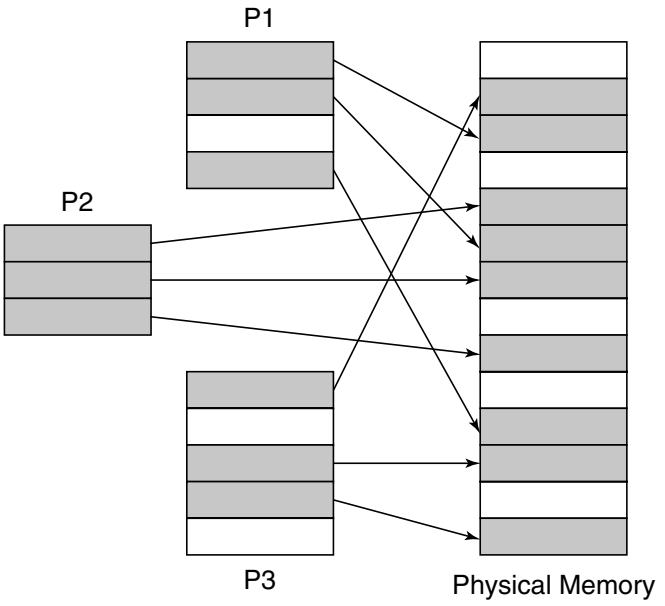


Fig. C.9 Page frames in the physical memory are assigned as pages to processes. All pages of a process need not be in the physical memory simultaneously

Whenever a process needs to access particular page which is not already loaded in the physical memory, a *page fault* occurs, the memory management function of the operating system takes over and loads the page from the disc into an empty page frame. In case an empty page frame is not available, a page frame which was not active for the largest length of time is emptied, by copying its contents to the swap disc if required (see Fig. C.10).

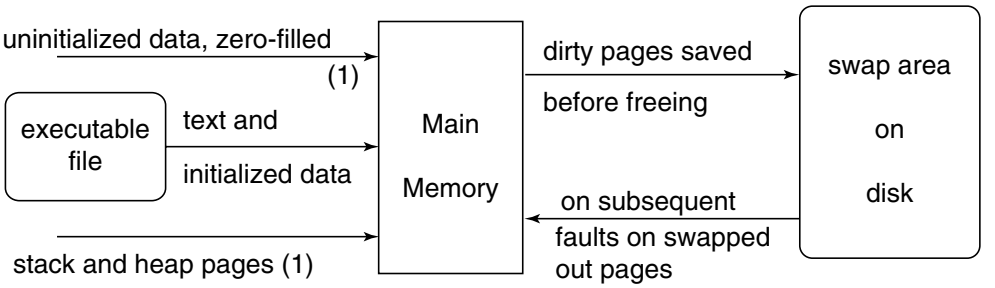


Fig. C.10 Virtual memory management

With such an arrangement, the memory assigned to all programs always begin at location 0. The libraries are usually shared between processes and only one copy is loaded as required in the physical memory. They have to use PIC, but the user (application) programs need not.

APPENDIX D

Worked-out Problems

D.1 Problems for Chapter 4: Parsers

D.1.1 Top-down Parsing

Problem 1.1.1 Is this a LL(1) grammar?

```
0. S' -> S#        3. A  ->  abS
1. S  -> aAa       4. A  ->  c
2. S  -> empty
```

First of all, we note that it is a CFG (single NT's on the LHS of all rules), it has ϵ-rule, there is no left recursion. So it is a candidate for being an LL(1) grammar. In FIRST(A), the RHS are disjoint, so we need test only productions from S. As one of the S productions is an ϵ-rule, we find the $FOLLOW(S) = \{\#,a\}$. Now, $FIRST(aAaFOLLOW(S)) \cap FIRST(\epsilon FOLLOW(S)) = \{a\} \cap \{\#, a\} \neq \varnothing$. Therefore, this is not an LL(1) grammar.

For example, consider a string "aaba… #". The productions for this strings can be: $S\# \rightarrow aAa\# \rightarrow aabSa \ldots \#$.

Now at this stage, we do not know whether to use rule 1 or rule 2 for the next substitution for S, looking at the look-ahead symbol 'a'.

Problem 1.1.2 Verify that the following grammar is LL(1).

```
0. S -> A#    3. B -> aB      5. C -> cC
1. A -> Bb    4. B -> empty   6. C -> empty
2. A -> Cd
```

We note that it is a CFG; there is no left recursion, but has ϵ-rules. So it is an LL(1) candidate grammar.

Verifying for:

A: there are no ϵ-rules, so we apply simple test:

$FIRST(Bb) \cap FIRST(Cd) = \{a, b\} \cap \{c, d\} = \varnothing$, OK.

B: $FOLLOW(B) = \{b\}$, $FIRST(aBb) \cap FIRST(b) = \varnothing$, OK.

C: $FOLLOW(C) = \{d\}$, $FIRST(cCd) \cap FIRST(d) = \varnothing$, OK.

Therefore, the grammar is LL(1).

Problem 1.1.3 Obtain the parsing function for the following LL(1) grammar with ϵ-rules.

```
0. S' -> S#        3. A  -> c
1. S  -> aSA       4. A  -> empty
2. S  -> empty
```

$FOLLOW(S) = \{c, \#\}$, $FOLLOW(A) = \{\#\}$

$FIRST(aSA) = \{a\}$, $FIRST(c) = \{c\}$

Verifying for:

A: $FIRST(c) \cap FOLLOW\ (A) = \{c\} \cap \{\#\} = \varnothing$, OK.

S: $FIRST(aSA) \cap FOLLOW(S) = \{a\} \cap \{c, \#\} = \varnothing$, OK.

The parsing function is given in Table D.1.

Table D.1 Parsing function for grammar in Problem 3

M	***a***	***c***	**#**
S	$(aSA,1)$	$(\epsilon,2)$	$(\epsilon,2)$
A		$(c,3)$	$(\epsilon,4)$
a	pop		
c		pop	
#			Accept

Problem 1.1.4 Verify that the following grammar is LL(1) and prepare the table of parsing function.

```
1. S -> aABC     4. B -> a        7. C -> empty
2. A -> a        5. B -> empty    8. D -> c
3. A -> bbD      6. C -> b        9. D -> empty
```

Verifying for:

A: $FIRST(a) \cap FIRST(bbD) = \varnothing$, OK.

B: $FIRST(a) \cap FOLLOW(B) = \{a\} \cap FIRST(C) = \{a\} \cap \{\mathrm{b}\} = \varnothing$, OK.

C: $FIRST(b) \cap FOLLOW(C) = \{b\} \cap \{\#\} = \varnothing$, OK.

D: $FIRST(c) \cap FOLLOW(D) = \{c\} \cap \{\#, b, a\} = \varnothing$, OK.

Therefore, it is an LL(1) grammar.

The parsing function is given in Table D.2.

Table D.2 Parsing function for grammar in Problem 4

M	***a***	***b***	***c***	**#**
S	$(aABC,1)$			
A	$(a,2)$	$(bbD,3)$		
B	$(a,4)$	$(\epsilon,5)$		$(\epsilon,5)$
C		$(b,6)$		$(\epsilon,7)$
D		$(\epsilon,9)$	$(c,8)$	$(\epsilon,9)$
a	pop			
b		pop		
c			pop	
#				Accept

Problem 1.1.5 Is the following grammar LL(1)? If so, build the parsing function.

```
1. S -> aSA        3. A -> bS
2. S -> empty      4. A -> c
```

It is a CFG, no left recursion, but an ϵ-rule, so possibly an LL(1) grammar.

$FOLLOW(S) = \{b,c\}$.

Testing for:

S: $FIRST(aSA) \cap FOLLOW(S) = \varnothing$, OK.

A: $FIRST(bS) \cap FIRST(c) = \varnothing$, OK.

Therefore, it is an LL(1) grammar.

The parsing function is given in Table D.3.

Table D.3 Parsing function for Problem 5

M	***a***	***b***	***c***
S	$(aSA,1)$	$(\epsilon,2)$	$(\epsilon,2)$
A		$(bS,3)$	$(c,4)$
a	pop		
b		pop	
c			Accept

Problem 1.1.6 Verify that the following grammar is LL(1).

```
1. V -> S#     4. A -> empty
2. S -> AB     5. B -> b
3. A -> a      6. B -> empty
```

It is a CFG, no left recursion, with ϵ-rules, so possibly an LL(1) grammar.

$FOLLOW(A) = \{b,\#\}$, $FOLLOW(B) = \{\#\}$. Verifying for:

A: $FIRST(a) \cap FOLLOW\ (A) = \varnothing$, OK.

B: $FIRST(b) \cap FOLLOW(B) = \varnothing$, OK.

Therefore, the grammar is LL(1).

Problem 1.1.7 Consider the following grammar RE over $\{a, b\}$, with union '+' and star '*' operations.

```
1. E   -> TE'        7. F  -> PF'     13. P -> empty
2. E'  -> +E         8. F' -> *F'
3. E'  -> empty      9. F' -> empty
4. T   -> FT'       10. P  -> (E)
5. T'  -> T         11. P  -> a
6. T'  -> empty     12. P  -> b
```

Compute the FIRST and FOLLOW sets. Then show that the grammar is LL(1).
Construct a parsing table, or, convert for a RDP and construct RDP.

$FIRST(TE') = \{(, a, b, +, *, empty\}$, $FIRST(FT') = \{(, a, b, *, empty\}$
$FOLLOW(E) = \{)\}$, $FOLLOW(E') = \{)\}$, $FOLLOW(T') = \{), +\}$,
$FOLLOW(F') = \{(, a, b, empty\}$,
Testing for LL(1):

For E': $FIRST(+E) \cap FOLLOW(E') = \varnothing$, OK.

For T': $FIRST(T) \cap FOLLOW(T') = \{(, a, b, *, empty\} \cap \{), +\} = \varnothing$, OK.

For F': $FIRST(*F') \cap FOLLOW(F') = \varnothing$, OK.

Therefore, the grammar is an LL(1).
Parsing table:

Table D.4 Parsing table for grammar in Example D.1.7

NT	+	*	(	)	*a*	*b*	ϵ
E			1		1	1	1
E'	2			3			
T			4		4	4	4
T'	6		5	6	5	5	5
F			7		7	7	7
F'		8	9		9	9	9
P			10		11	12	13

Conversion of the grammar for RDP: we have to convert it to extended BNF.

```
1. E -> T {+T}      4. P -> (E) | a | b | empty
2. T -> F {F}
3. F -> P {*P}
```

A skeleton RDP for the grammar:

```
pE(){pT(); while(next() == '+'){pplus(); pT();}}
pT(){pF(); while(strchr(next(),"(ab+"){pF();}}
pF(){pP(); while(next() == '*'{past();pP();}}
pP(){if(next() == '('){plp(); pE(); prp();}
     else if(next() == 'a') pa();
     else if(next() == 'b') pb();
     else error("unexpected token");
}
pplus(){if(symbol() != '+') error("+ expected");}
past(){if(symbol() != '*') error("* expected");}
plp(){if(symbol() != '(') error("( expected");}
prp(){if(symbol() != ')') error(") expected");}
pa(){if(symbol() != 'a') error("a or b expected");}
pb(){if(symbol() != 'b') error("a or b expected");}
```

D.1.2 Bottom-up Parsing

Problem 1.2.8 Consider the following grammar defining a list structure:

```
1. S -> a     4. T -> T,S
2. S -> n     5. T -> S
3. S -> (T)
```

Here '*n*' stands for an empty list. An empty list is still a list, different from an empty production.

Give leftmost and rightmost derivations for `(a,(a,a))` and indicate the handle at each stage in the rightmost derivation. Then develop an LR(0) machine for this grammar. Identify inadequate states, if any, and check if SLR(1) resolution is possible.

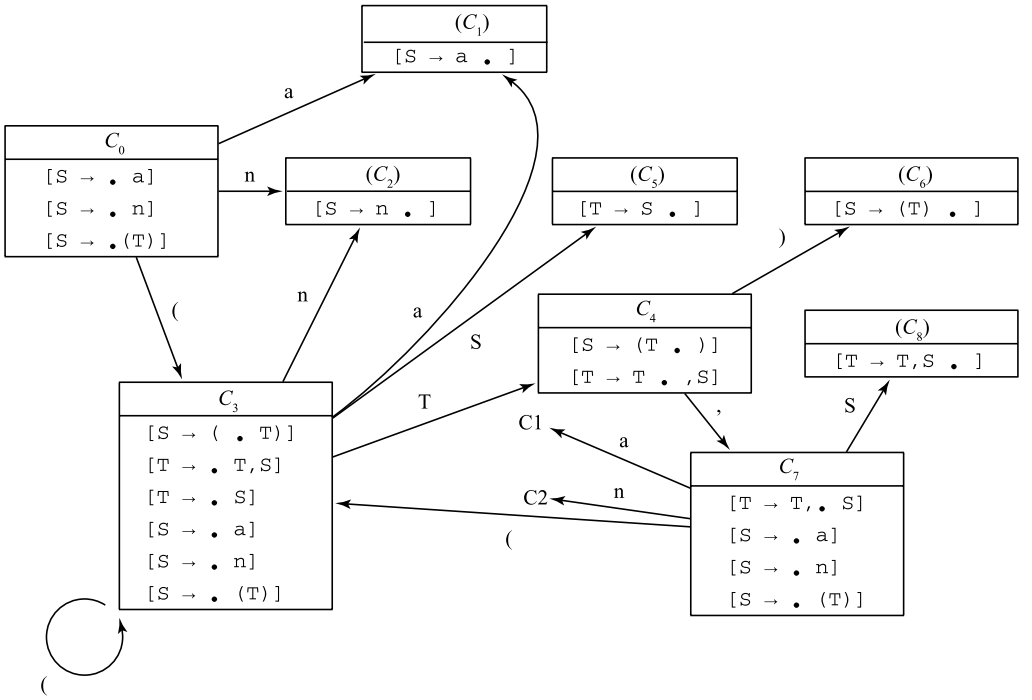
Leftmost derivation:

$S \to (T) \to (T, S) \to (S, S) \to (a, S) \to (a, (T)) \to (a, (T, S)) \to (a, (S, S)) \to (a, (a, S)) \to (a, (a, a))$

Rightmost derivation – handles are underlined:

$S \to (\underline{T}) \to (\underline{T, S}) \to (T, (\underline{T})) \to (T,(\underline{T, S})) \to (T,(T, \underline{a})) \to (T,(\underline{S}, a)) \to (T, (\underline{a}, a)) \to (\underline{S}, (a, a)) \to (\underline{a}, (a, a))$

The LR(0) machine is:



Seemingly, there are no inadequate states.

Problem 1.2.9 For the grammar given below, find the viable prefixes for the sentential forms `E + a * a#` and `E + P^(a + a)#`.

```
1. S -> E#          4. T -> P          7. P -> P * F
2. E -> T           5. T -> P ^ T      8. F -> a
3. E -> E + T       6. P -> F          9. F -> ( E )
```

For the sentential form `E + a * a#`: the derivation is:
$E + F * a\# \Rightarrow \underbrace{E + a}_{\phi\beta} * a\#$, then the viable prefixes are: E, $E+$, $E + a$.

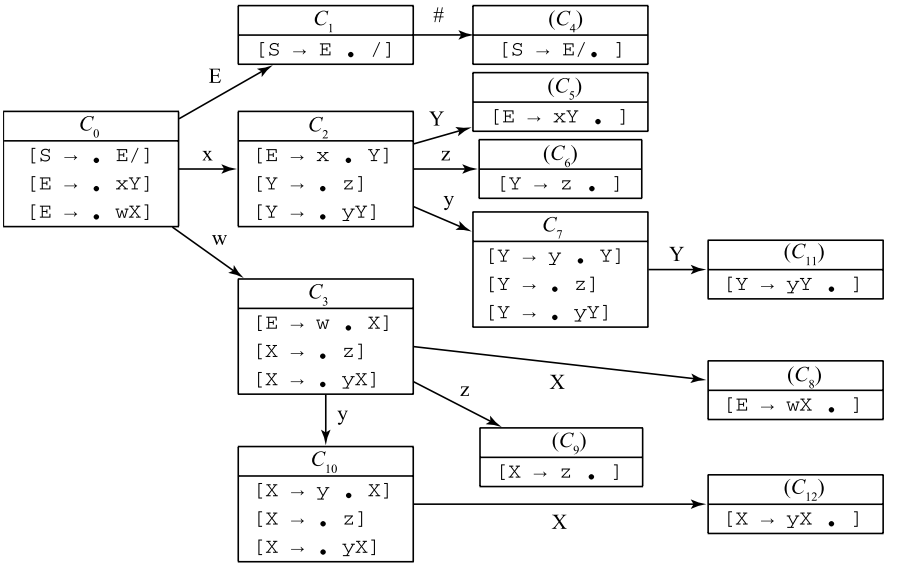
For sentential form `E + P ^ (a + a)#`: The derivation is: $E + P\,\hat{}\,(F + a)\# \Rightarrow \underbrace{E + P\,\hat{}\,(a}_{\phi\beta} + a)\#$, then the viable prefixes are: E, $E+$, $E + P$, $E + P$^, $E + P$^(, $E + P$^(a.

Problem 1.2.10 Consider the grammar:

```
1. S -> E#    4. X -> yX    6. Y -> yY
2. E -> wX    5. X -> z     7. Y -> z
3. E -> xY
```

Compute the LR(0) machine. Does it have any inadequate states? Show how will an example string `wyyz#` be parsed.

The LR(0) machine is:



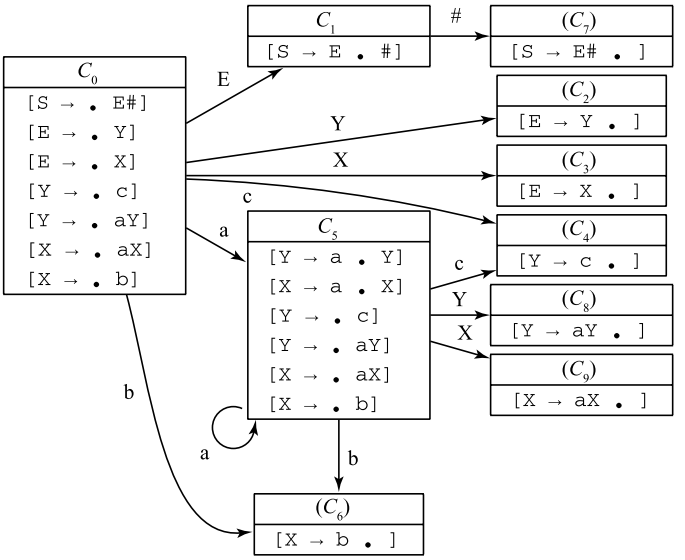
The machine does not have any inadequate states. The parsing and acceptance of the example string `wyyz#` is shown in Table D.5.

Table D.5 Parsing of an example string `wyyz#`

Stack	Input	Handle, rule
0	*wyyz#*	
0 *w*3	*yyz#*	
0 *w*3 *y*7	*yz#*	
0 w3 *y*7 *y*7	*z#*	
0 *w*3 *y*7 *y*7 *z*8	#	$X \rightarrow z$, 5
0 *w*3 *y*7 *y*7 *X*9	#	$X \rightarrow yX$, 4
0 *w*3 *y*7 *X*9	#	$X \rightarrow yX$, 4
0 *w*3 *X*4	#	$E \rightarrow wX$
0 *E*1	#	
0 *E*1 #		$S \rightarrow E\#$
0 *S*		Accept

Problem 1.2.11 Develop an LR(0) machine for the following grammar. Does it have any inadequate states?

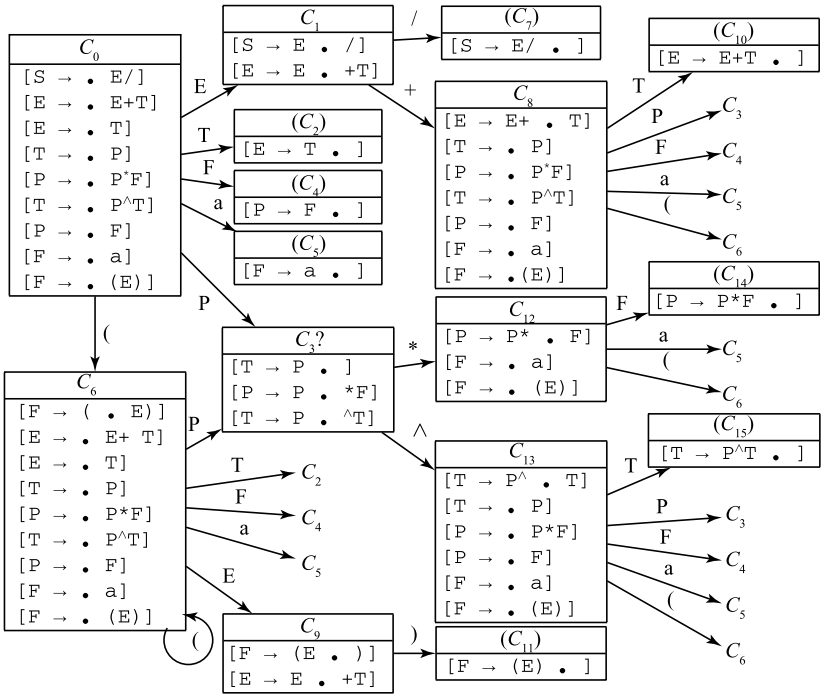
```
1. S -> E#    4. X -> aX    7. Y -> c
2. E -> X     5. X -> b
3. E -> Y     6. Y -> aY
```



From the LR(0) machine, it is seen that there are no inadequate states.

Problem 1.2.12 For the grammar given in Problem 9, compute the LR(0) machine, show that it is an SLR(1) grammar, prepare an SLR(1) parsing table, and trace of parse of `a + a ^ (a + a) #`.

The LR(0) machine is:



There is only one inadequate state 3, where we have Shift/Reduce conflict – shift on '*' and '^', reduce to *T*.

To show that this is an SLR(1) grammar, we have to construct the *FOLLOW*(*T*) set.

$$FOLLOW(T) = \{+,), /\}$$

which is disjoint with respect to {*, ^}, so we have an SLR(1) grammar.

To compute the *f*() and *g*() functions, we need the following FOLLOW sets:

FOLLOW(*E*) = {+,), /}, while reducing by rules 2, 3;

FOLLOW (*T*) = {+,), /}, while reducing by rules 4, 5;

FOLLOW(*P*) = {*, ^+,), /}, while reducing by rules 6, 7;

FOLLOW(*F*) = {*, ^+,), /}, while reducing by rules 8, 9.

The SLR(1) parse functions are given in Table D.6.

Table D.6 Parsing functions *f*( ) and *g*( ) for Problem 12

State	+	*	^	(	)	*a*	/	*S*	*E*	*T*	*P*	*F*	+	*	^	(	)	*a*	/
0				*S*		*S*			1	2	3	4				6		5	
1	*S*						*S*						8						7
2	*r*2																		
3	*r*4	*S*	*S*		*r*4		*r*4							12	13				
4	*r*6	*r*6	*r*6		*r*6		*r*6												
5	*r*8	*r*8	*r*8		*r*8		*r*8												
6				*S*		*S*			9	2	3	4				6		5	
7							*r*1												
8				*S*		*S*				10	3	4				6		5	
9	*S*				*S*								8				11		
10	*r*3				*r*3		*r*3												
11	*r*9	*r*9	*r*9		*r*9		*r*9												
12				*S*		*S*						14				6		5	
13										15	3	4				6		5	
14	*r*7	*r*7	*r*7		*r*7		*r*7												
15	*r*5				*r*5		*r*5												

Table D.7 shows the trace of parse of an input string `a + a ^ (a + a) /`

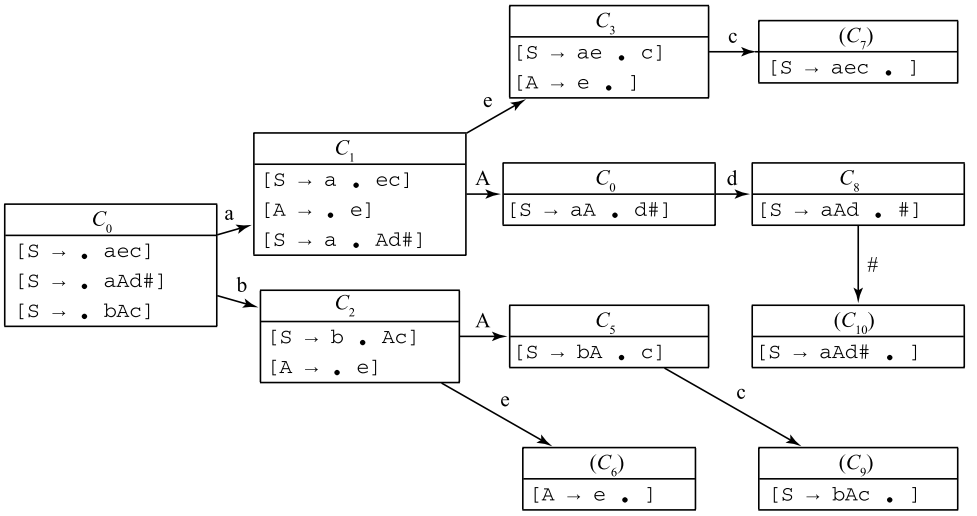
Table D.7 Trace of parse of `a + a ^ (a + a) /` – Problem 12

Stack	Input	Action and rule
0	a+a^(a+a)/	Shift
0 *a*5	+a^(a+a)/	Reduce by 8
0 *F*4	+a^(a+a)/	Reduce by 6
0 *P*3	+a^(a+a)/	Reduce by 4
0 *T*2	+a^(a+a)/	Reduce by 2
0 *E*1	+a^(a+a)/	Shift
0 *E*1 +8	a^(a+a)/	Shift
0 *E*1 +8 *a*5	^(a+a)/	Reduce by 8
0 *E*1 +8 *F*4	^(a+a)/	Reduce by 6
0 *E*1 +8 *P*3	^(a+a)/	Shift
0 *E*1 +8 *P*3 ^13	(a+a)/	Shift
0 *E*1 +8 *P*3 ^13 (6	a+a)/	Shift
0 *E*1 +8 *P*3 ^13 (6 *a*5	+a)/	Reduce by 8
0 *E*1 +8 *P*3 ^13 (6 *F*4	+a)/	Reduce by 8
0 *E*1 +8 *P*3 ^13 (6 *P*3	+a)/	Reduce by 8
0 *E*1 +8 *P*3 ^13 (6 *T*2	+a)/	Reduce by 2
0 *E*1 +8 *P*3 ^13 (6 *E*9	+a)/	Shift
0 *E*1 +8 *P*3 ^13 (6 *E*9 +8	a)/	Shift
0 *E*1 +8 *P*3 ^13 (6 *E*9 +8 *a*5	)/	Reduce by 8
0 *E*1 +8 *P*3 ^13 (6 *E*9 +8 *F*4	)/	Reduce by 6
0 *E*1 +8 *P*3 ^13 (6 *E*9 +8 *P*3	)/	Reduce by 4
0 *E*1 +8 *P*3 ^13 (6 *E*9 +8 *T*10	)/	Reduce by 3
0 *E*1 +8 *P*3 ^13 (6 *E*9	)/	Shift
0 *E*1 +8 *P*3 ^13 (6 *E*9)11	/	Reduce by 9
0 *E*1 +8 *P*3 ^13 *F*4	/	Reduce by 6
0 *E*1 +8 *P*3 ^13 *T*15	/	Reduce by 5
0 *E*1 +8 *T*10	/	Reduce by 3
0 *E*1	/	Shift
0 *E*1 /7		Reduce
0 *S*		Accept

Problem 1.2.13 Consider the following grammar.

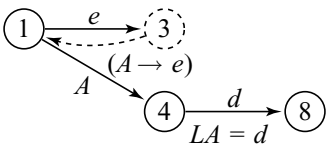
```
1. S -> aAd#
2. S -> bAc
3. S -> aec
4. A -> e
```

Obtain it's LR(0) machine and LALR(1) LA sets for the inadequate states if any.



The only inadequate state is 3, which has Shift/Reduce conflict. We find that $FOLLOW(A) \cap \{c\} = \{d, c\} \cap \{c\} \neq \varnothing$.

Therefore, SLR(1) technique cannot be used. We should then find the look-ahead set (LA) for A when the FSM is in state 3, $LA(3, A) = \{d\}$.

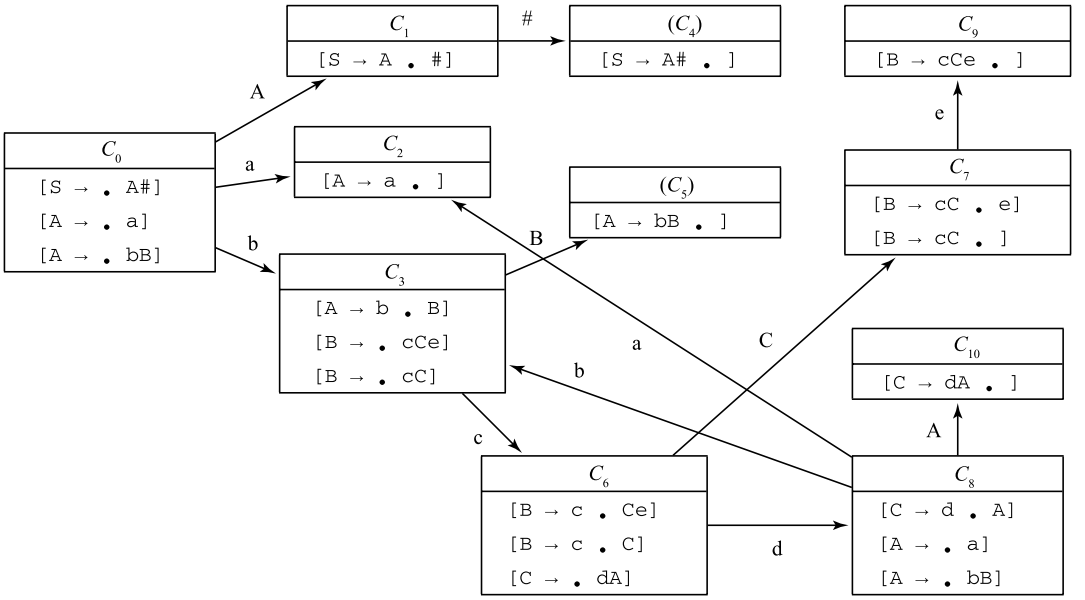


This gives us $LA(3, A) \cap \{c\} = \{d\} \cap \{c\} = \varnothing$; thus, LALR(1) resolution is possible.

Problem 1.2.14 Consider the following grammar.

```
1. S -> A#     4. B -> cC
2. A -> bB     5. B -> cCe
3. A -> a      6. C -> dA
```

Is it an LALR(1) grammar? If so, obtain the LA sets for the inadequate states if any.



The only inadequate state is 7, with Shift/Reduce conflict on B. So we find

$$FOLLOW(B) = \{\#, e\} \text{ and } \{\#, e\} \cap \{e\} \neq \varnothing$$

so it is not an SLR(1) grammar.

Then in order to check if LASL(1) can be applied, we find LA(7,B). Suppose we have reduction at state 7, then two symbols 'C' nd 'c' are unstacked and state 3 is *uncovered*. On taking a transition on 'B' from here, we arrive at state 5, where another reduction takes place on 'b''B' and state 0 is uncovered. Taking transition 'A' from there we arrive at state 2, where the look-ahead symbol is #. Therefore LA(7,B) = {#} and as this is disjoint with $\{e\}$, we can apply LALR(1) resolution for the inadequate state 7.

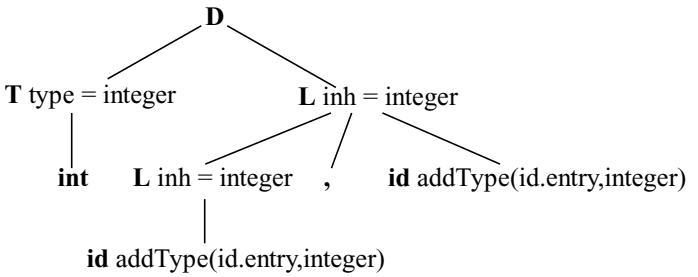
D.2 Problems for Chapter 5: Syntax-directed Translation

D.2.1 Syntax-directed Definition (SDD)

Problem 2.1.15 Given below the syntax-directed definition (SDD), construct the annotated parse tree for the input expression: "int a, b".

```
D -> T L        L.inh = T.type
T -> int        T.type = integer
T -> float      T.type = float
L -> L1, id     L1.inh = L.inh
                addType(id.entry,L.inh)
L -> id         addType(id.entry,L.inh)
```

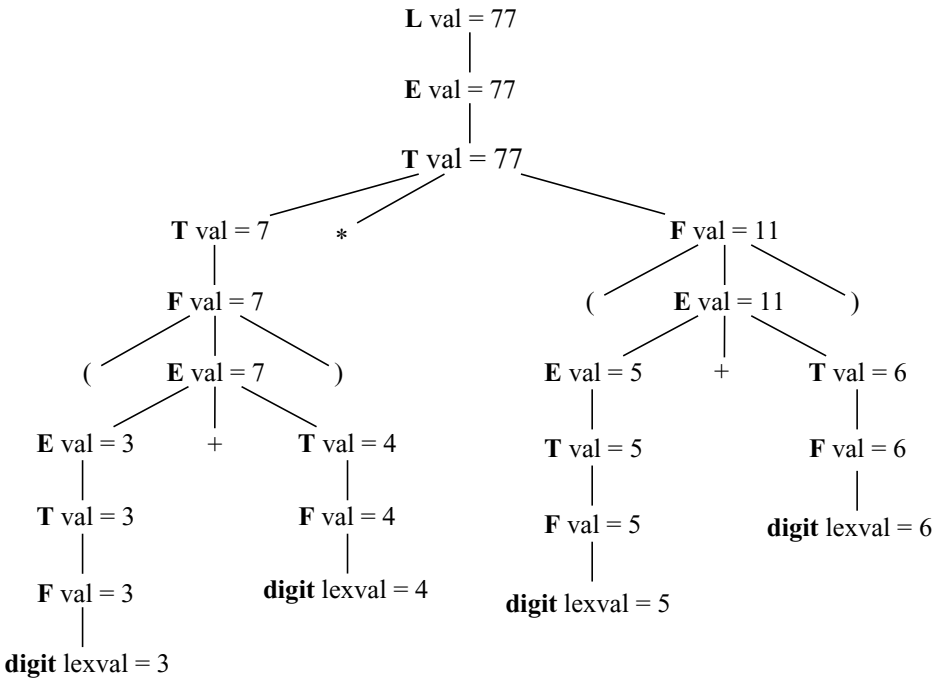
The annotated parse tree is:



Problem 2.1.16 Given below the SDD with the synthesized attribute `val`, draw the annotated parse tree for the expression (3+4)*(5+6).

```
L -> E          L.val = E.val
E -> T          E.val = T.val
E -> E1 + T     E.val = E1.val + T.val
T -> F          T.val = F.val
T -> T1 * F     T.val = T1.val * F.val
F -> ( E )      F.val = E.val
F -> digit      F.val = digit.lexval
```

The annotated parse tree is:



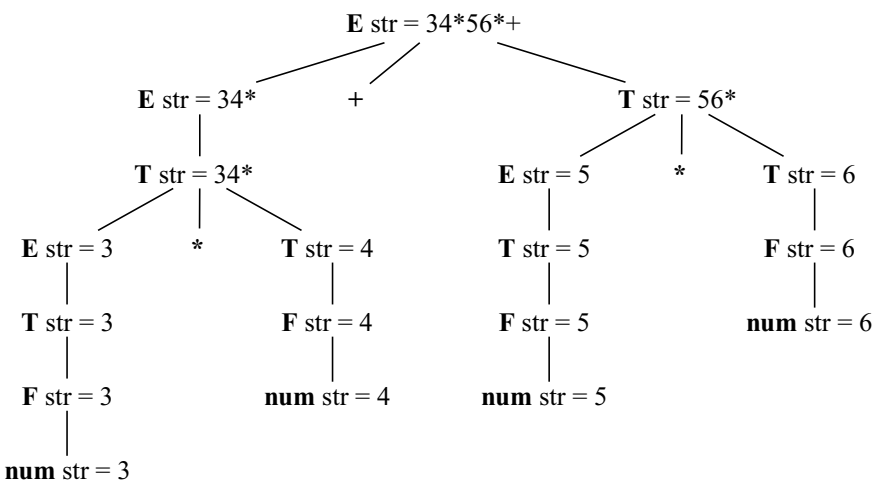
D.2.2 Syntax-directed Translation (SDT)

Problem 2.2.17 Prepare an SDT scheme that translates arithmetic expressions from *infix* into *postfix* notation. The solution should include the CFG, the semantic attributes for each of the grammar symbols and semantic rules. Show the application of your scheme to the input "3*4 + 5*2".

Solution: We define a synthesized attribute string and the string concatenation operator ':'. We also define a simple grammar as shown below with start symbol `E` and with Terminal symbol `num`, which also has a string attribute set by the lexical analyzer as the string with the numeric value representation of `num`.

```
E1 -> E2 + T      E1.str = E2.str : T.str : '+'
E1 -> T           E1.str = T.str
T1 -> T2 * F      T1.str = T2.str : F.str : '*'
T  -> F           T.str  = F.str
F  -> ( E )       F.str  = E.str
F  -> num         F.str  = num.str
```

For the input string "3*4+5*6", we would have the annotated parse tree shown below.



Problem 2.2.18 Consider the following grammar for expressions and lists of statements (`stmtlist`) using assignment statements (`assgn`) and basic expressions (`expr`) using the productions presented below.

```
stmtlist  -> stmt ; stmtlist
stmtlist  -> stmt
stmt      -> assgn
expr      -> expr + expr
expr      -> int
expr      -> id
assgn     -> id = expr
assgn     -> id += expr
```

Using a stack-based machine, write a syntax-directed definition to generate code for `stmtlist`, `assgn` and `expr`.

Solution: Remember that in a stack-based machine you need to push the operands of the expressions first onto the stack and then execute the operations with the topmost element/s of the stack.

A possible solution is to have only synthesized attributes such as "str" for identifiers and "val" for integers. The non-terminal `expr` also has a synthesized attribute that holds the variable name when defined by an identifier. When defined by an integer this attribute value is `null`.

A possible syntax-directed translation definition using the `yacc`-like assignment of non-terminal symbols in a production is shown below.

```
stmtlist ->  stmt ; stmtlist  { }
stmtlist ->  stmt             { }
stmt     ->  assgn            { }
expr     ->  expr + expr      { emit("add"); }
expr     ->  int              { emit("push int.val"); $$.val = nil; }
expr     ->  id               { emit("push id.str"); $$.val = id.str; }
assgn    ->  id = expr        { emit("pop $1.val"); }
assgn    ->  id += expr       { emit("add"); emit("push $1.val");
                                emit("pop $1.val");}
```

D.3 Problems for Chapter 6: Type Checking

D.3.1 Type as Attribute

In Pascal language, a programmer can declare two integer variables `a` and `b` with the syntax

```
var a,b: int
```

This declaration might be described with the following grammar:

```
decl   -> var idlist : typeid
idlist -> idlist , ID
idlist -> ID
```

Write an attribute grammar that assigns the correct data type to each declared variable. (You may have to rewrite the grammar.) Determine an evaluation order of the attributes irrespective of the way the parse tree is constructed. Can this translation be performed in a single pass over the parse tree as it is created?

Solution: The grammar would require an inherited attribute for passing the type of the identifier to the top of the parse tree where the `idlist` non-terminal is rooted. This inherited attribute causes all sort of problems, the important one being that it would preclude the evaluation in a single pass if the tree were created using a bottom-up and left-to-right parsing approach.

The best way would be to change the grammar as indicated below, where all the attributes would be synthesized and hence allow the evaluation of the attribute in a single pass using the bottom-up left-to-right parsing approach.

```
decl  ->  var ID list            ID.type    = list.type
list0 ->  , ID list1             ID.type    = list0.type
                                 list1.type = list0.type
list0 ->  : typeid               list0.type = typeid.value
```

D.4 Problems for Chapter 7: Run-Time Environment

D.4.1 Storage Allocation

Problem 4.1.19 We have seen a few language constructs – like variable length strings, dynamic arrays, nested and recursive procedures, etc. – that cannot be handled by static storage allocation. What other constructs cannot be handled by static storage allocation?

Solution:

1. Explicit storage request and release (similar to `malloc()` and `free()` in C) at run-time.
2. Run-time output format specifications, for example, as available in PL/I.

Problem 4.1.20 It is suggested that the value of variables be kept in the symbol table during run-time (as it was done in the original BASIC language implementations). Discuss advantages and disadvantages of this scheme. Are there any implementation issues?

Solution:

Advantages: (i) During program debugging, value of the variable is accessible by an identifier. Also, the same thing can be done during run-time. For example, an `eval()` function can be implemented, which allows the user to give a symbolic expression at run-time and then it is evaluated and the result returned. (ii) Run-time modifications of a variable value, especially in a debugger (like `gdb`), to test the effect of changing a variable value on running of the program. (iii) A small reduction in memory requirements, as the symbol table need not store the address of the variable.

Disadvantages: (i) The complete symbol table has to be retained during run-time. (ii) Data types are restricted to only those few, which occupy a reasonably modest amount of memory. (iii) Some waste of space – integers are also stored as reals (as was done in BASIC).

Implementation issues: How would we store (i) arrays, (ii) strings, (iii) structures?

Problem 4.1.21 Justify: "Dynamic storage allocation strategy is particularly suitable for run-time requirements of a block-structured language".

Solution: A block-structured language implies a Last-In-First-Out kind of behaviour due to its nested execution structure. This matches well with the stack-oriented storage allocation in dynamic storage allocation strategy.

D.4.2 Call Graph

Problem 4.2.22 Consider the following C program:

```
void output(int n, int x){
  printf("The value of %d! is %d.\n",n,x);
}
int fact(int n){
  int x;
  if(n > 1) x = n * fact(n - 1);
  else x = 1;
  output(n,x);
  return x;
}
```

```
void main(){
  fact(4);
}
```

Show its *call graph*, i.e. caller–callee relationship for user-defined functions. Show its *call tree* and its execution history, i.e. the arguments' values and output produced. Discuss for this particular section of the code if the Activation Records (AR) can be allocated statically or not. Explain why or why not.

Solution: The call graph and call tree are shown in Fig. D.1.

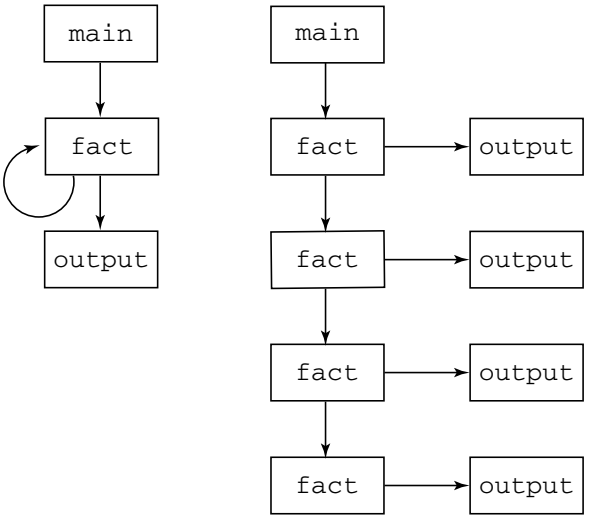


Fig. D.1 Call graph and call tree

The execution history is:

```
main      calls     fact(4)
fact(4)   calls     fact(3)
fact(3)   calls     fact(2)
fact(2)   calls     fact(1)
fact(1)   calls     output
output    returns   to fact(1)
fact(1)   returns   to fact(2)
fact(2)   calls     output
output    returns   to fact(2)
fact(2)   returns   to fact(3)
fact(3)   calls     output
output    returns   to fact(3)
fact(3)   returns   to fact(4)
fact(4)   calls     output
output    returns   to fact(4)
fact(4)   returns   to main
```

In general, the ARs cannot be allocated statically whenever they involve functions that are either directly recursive as it the case of `fact` or mutually recursive. In the present case, there a cycle in the call graph revealing that `fact` is recursive. As a result, the AR for `fact` cannot be allocated statically. However, the frames for `output` can be allocated statically as there is a single active instance of output at a given point in time.

Problem 4.2.23 Consider the following C program:

```
int table[1024];
void Output(int n, int x){
  printf(" Fib of %d is %d\n",n,x);
}
void fillTable(int idx){
  int i;
  for(i = 2; i <= idx; i++){
    table[i] = table[i - 1] + table[i - 2];
  }
}
int fib(int idx){
  if(table[idx] == 0){
    fillTable(idx);
  }
  return table[idx];
}
int main(int argc, char ** argv){
  int idx, n, k;
  for(k = 0; k < 1024; k++)
      table[k] = 0;
  table[0] = 1;
  table[1] = 1;
  while(1){
    scanf("%d",&n);
    if((n <= 0) || (n >= 1024))
      exit(0);
    k = fib(n);
    Output(n,k);
  }
}
```

Show its call graph, i.e. caller–callee relationship for user-defined functions.

Show its execution history, i.e. the arguments' values and output produced when you input the value '4' and then you input the value '3' and lastly the value '0'.

Discuss for this particular section of the code if the AR can be allocated statically or not. Explain why or why not.

Solution: The call graph is given in Fig. D.2:

The execution history is:

```
main()
scanf("%d",&n)
fib(4)
fillTable(4)
Output(4,5)
printf(" Fib of %d is %d\n",4,5);
scanf("%d",&n)
fib(3)
Output(3,3)
printf(" Fib of %d is %d\n",3,3);
```

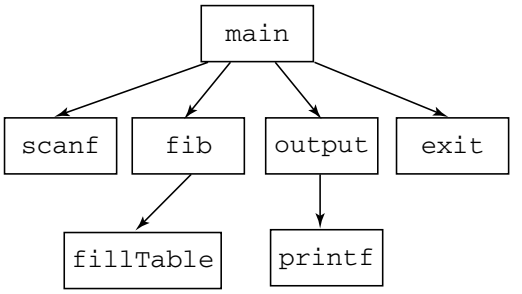


Fig. D.2 Call graph – note that there is no cycle

Notice that `fillTable()` is not called after `fib(3)`.

In this case, there is no cycle in the call graph revealing that `fib` is not recursive. As a result, the AR for fib can be allocated statically.

D.4.3 Function Storage and Activation Record

Problem 4.3.24 Consider the following C program and assume that the ARs follow the same general layout as standard Cdecl layout. Assuming that an integer requires 4-bytes and a double data type 8-bytes, derive a layout for the local variables for the AR of the enclosing procedure *P*.

```
B0: {
    int a, b, c
    /* ... assign value to a and b */
B1:   {
          int v(a), d, w(b)
          double x;
B2:       {
            int x, y(8)
            /* .... */
            }
      }
B3:   {
            int f, g;
            /* .... */
      }
    }
```

Solution: The AR is shown in Fig. D.3. Note the sharing of the stack space between pointer to `v` and `f`, `g` and `d`.

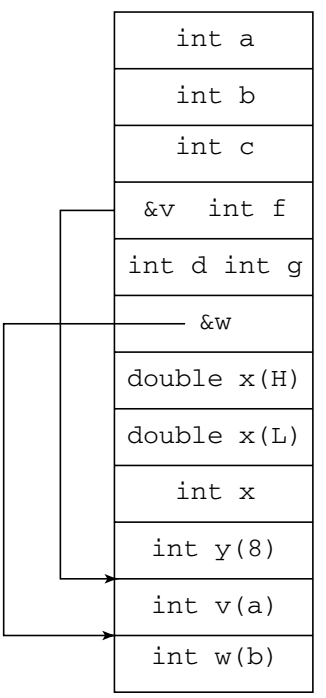


Fig. D.3 AR showing local variables

This page is intentionally left blank.

Bibliography

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2002.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[App98b] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[App02] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[CT04] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[Dha99] D. M. Dhamdhere. *Systems Programming and Operating Systems*. Tata McGraw-Hill, 1999.

[Gri71] David Gries. *Compiler Construction for Digital Computers.* John Wiley and Sons, New York, 1971.

[Han85] Per Brinch Hansen. *Brinch Hansen on Pascal Compilers*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[HF95] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley, 1995.

[HMU00] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Pearson Education Asia, India, 2000.

[Hol90] A. I. Holub. *Compiler Design in C.* Prentice-Hall of India, India, 1990.

[Joh75] Stephen C Johnson. Yacc: Yet another compiler compiler. computing science technical report no. 32. Technical report, Bell Laboratories, Murray hill, New Jersey, 1975.

[Knu65] D.E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, October 1965.

[KP87] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall of India, New Delhi, 1987.

[LB92] Tony Mason Levine, John R. and Doug Brown. *Lex and Yacc*. O'Reilly and Associates, Inc., Sebastopol, California, 1992.

[LS75] M. E. Lesk and E. Schmidt. Lex a lexical analyzer generator. computing science technical report no. 39. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Mar03] J. Martin. *Introduction to Languages and Theory of Computation*. Tata McGraw-Hill Publishing Co Ltd, India, 2003.

[Mor98a] C. Robert Morgan. *Building an Optimizing Compiler*. Butterworth, 1998.

[Mor98b] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. Thomson Learning, Boston, 1997.

[SS02] Y. N. Srikant and Priti Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[WC93] William M. Waite and Lynn Robert Carter. *An Introduction to Compiler Contruction*. HarperCollins, 1993.

Index

A

Absolute
-Address, 308
-Loader, 3, 456
-Value, 242, 308, 329
Abstract
-Mathematics, 393
-Syntax, 37, 227, 239, 240, 265, 270
-Tree, 51
Acceptor, 10, 11, 15, 50, 75, 125, 412, 415, 417, 422–424
Access
Array-, 308
Base-, 347
Database-, 347
Direct-, 334
Dynamically-, 222
Element-, 221
Fast-, 58, 160, 188
Faster-, 289
File-, 219
Full-, 434
Generator-, 270
Memory-, 160, 276, 278, 290–292, 295, 308, 309, 346
Operand-, 309
Place-, 188
Table-, 214, 348, 384
Time-, 219, 278
Units-, 274
Unix-, 226
Utime-, 219
Action
-Associated, 65, 66, 257
-Code, 153, 159, 249
-Disappear, 128
-Entries, 100
-Function, 100
-Mathematics, 393
-Mechanism, 283
-Required, 220
-Right, 387
-Specified, 116, 123, 125
-Statements, 384
-Suitable, 61
-Symbols, 152
-Synthesizes, 158
-Taken, 73, 176, 418
-Term, 151, 152, 154, 161–164, 249, 253, 255, 351, 367–372, 384
-Terms, 151, 152, 161–164, 249, 253, 351, 367–372, 384
Actual-, 384
Associate-, 119
Automatic-, 114
Codegen-, 43, 44
Complete-, 24
Complicated-, 123
Correct-, 104
Default-, 65, 66
ECHO-, 66
Embedded-, 153
Exact-, 140
FPU-, 387
Generating-, 161
Hardware-, 323
Implied-, 143, 306
Included-, 54
Initialization-, 251
Next-, 95
Optimization-, 291
Optional-, 65
Outstanding-, 128
Parser-, 95, 96
Parsing-, 82
Practical-, 335
Previous-, 120
Program-, 205
Proper-, 161, 164
Reduce-, 104, 124
Required-, 164
Rules-, 153
Semantic-, 128, 133, 160, 165
Setting-, 161
Shift-, 104
Specified-, 267
Unit-, 387
Widespread-, 283
Yacc-, 178, 179, 351, 367–369, 371

Actions
Associate-, 119
Automatic-, 114
Codegen-, 43, 44
Complicated-, 123
Embedded-, 153
FPU-, 387
Generating-, 161
Implied-, 143, 306
Included-, 54
Initialization-, 251
Optimization-, 291
Outstanding-, 128
Parser-, 95, 96
Previous-, 120
Program-, 205
Reduce-, 104, 124
Semantic-, 128, 160, 165
Shift-, 104
Specified-, 267
Unit-, 387
Yacc-, 178, 179
Activation
-Record, 182, 195, 196, 243, 265, 287, 476, 478
-Records, 182, 196, 243, 476
Address
-Adjustments, 192, 451
-Arithmetic, 3, 434
-Array, 42
-Assigned, 38
-Assignment, 270
-Base, 287
-Calculation, 160, 199, 280, 281, 435
-Calculations, 199, 280, 435
-Code, 37–40, 238, 324
-Constant, 187, 436, 450, 453, 456–458
-Constants, 187, 436, 450, 456, 457
-Contained, 309
-Counters, 42
-Due, 457
-Equivalents, 436
-Esp, 284
-Generation, 287
-Instructions, 49, 241, 270
-Intermediate, 41, 42
-Involves, 48
-Linkage, 220
-Links, 25
-Loader, 42
-Memory, 390
-Mode, 30, 42, 47, 113, 270, 275, 308, 457
-Modes, 42, 47, 113, 270, 275, 308, 457
-Must, 457
-Negative, 190
-Operand, 276, 456, 457
-Operands, 456, 457
-Operations, 256
-Positive, 190
-Reference, 197
-Relative, 186, 309
-Returned, 203
-Scheme, 199
-Sent, 457
-Space, 455
-Stack, 42
-Stored, 188
-Technique, 308
-Used, 42
-Value, 437, 453
-Virtual, 238
Absolute-, 160, 186, 222, 308
Arbitrary-, 279, 333
Area-, 203
Array-, 280, 281, 287, 324
Assigned-, 32, 453
Base-, 48, 160
Based-, 287
Beginning-, 457
Call-, 349
Code-, 187, 202, 247, 266
Complex-, 308
Component-, 199, 287, 457
Computed-, 354
Contains-, 375
Dependent-, 457
Destination-, 357
Direct-, 30, 246, 309, 378, 381
Effective-, 309, 382, 457
Element-, 221
Equivalent-, 437
Even-, 456
Example-, 187
Express-, 3
Field-, 266
Final-, 187, 270, 309
Immediate-, 309, 390
Implied-, 309
Increment-, 308
Indexed-, 48, 284, 309
Indirect-, 246, 309
Instruction-, 43
Jump-, 42, 43, 450
Line-, 256
Load-, 3, 186, 187, 450, 453, 456–458
Memory-, 3, 42, 43, 151, 204, 225, 238, 270, 276, 278, 290, 308, 354, 390, 448, 455–457
Mode-, 381, 390
Operand-, 308, 457
PIC-, 457
Pointer-, 381, 382
Proper-, 42, 220
Provide-, 434, 436
Random-, 42
Relative-, 160, 186, 222, 309, 457
Required-, 451
Return-, 187–191, 193, 195, 196, 242, 243, 283, 285–288
Saved-, 43, 287
Specified-, 267, 380, 442
Stack-, 287, 372
Start-, 42, 188, 456, 457

Starting-, 450
Storage-, 163
Subroutine-, 245
Symbolic-, 434, 436
Term-, 287
Time-, 199
Unique-, 248
Virtual-, 222
Zero-, 42
Addresses
-Must, 457
Absolute-, 160, 186, 222
Arbitrary-, 333
Array-, 280
Call-, 349
Code-, 187, 247
Dependent-, 457
Equivalent-, 437
Express-, 3
Final-, 187
Jump-, 43, 450
Load-, 450, 457
Memory-, 3, 238, 278, 390, 456, 457
Operand-, 308
Relative-, 160, 186, 222, 457
Return-, 286
Saved-, 287
Subroutine-, 245
Symbolic-, 434, 436
Virtual-, 222
Ahead
Int-, 57
Look-, 57, 74, 103, 106, 367
Return-, 57
Aho, 137, 165, 181, 225, 331
Algorithm
Algorithms-, 2, 412
Developing-, 87
Discussing-, 239
Extended-, 39
Finding-, 412
Fit-, 204
General-, 404
Good-, 269
Inefficient-, 329
Known-, 331
Optimization-, 311, 329
Parser-, 106, 367
Parsing-, 50, 75, 89, 91, 127
Particular-, 329
Recursive-, 406
Scanner-, 33
Step-, 4
System-, 204
Underlying-, 5
Algorithms
Developing-, 87
Discussing-, 239
Finding-, 412
Optimization-, 311, 329
Parser-, 367
Parsing-, 89
Recursive-, 406
Step-, 4
Underlying-, 5
Allocation
-Arising, 191
-Code, 188
-Function, 434
-Managed, 182
-Method, 185, 188, 191, 202
-Methods, 185, 188, 191
-Occurs, 203
-Requests, 186, 451
-Scheme, 186
-Strategy, 190, 202, 475
-Words, 346
Area-, 42, 451
Based-, 191
Demand-, 191
Dynamic-, 190
Explicit-, 203
Heap-, 182, 202
Implementation-, 203
Memory-, 160, 182, 185–188, 191, 197, 202, 203, 266, 334, 346, 434, 456
Proper-, 274
Register-, 19, 268, 272, 278, 286, 292, 300, 325
Allocation
Request-, 455
Static-, 186
Storage-, 114, 182, 185, 186, 193, 202, 209, 475
Time-, 191
Ambiguity
-Example, 425
Avoid-, 395
Local-, 105
Trivial-, 425
Analysis
lexical, 11, 13, 50–52, 75
syntax, 11, 13, 15, 51, 74, 75, 140
Expression-, 326
Flow-, 325
Lexical-, 13, 51
Live-, 326, 327
Model-, 393
Semantic-, 13, 51, 329
Syntax-, 13, 51
Variable-, 326
Analyzer
lexical, 13, 14, 25, 26, 47, 48, 50, 51, 53, 55, 57, 59, 61–63, 65, 67–69, 71, 73, 114, 115, 120, 124, 125, 154, 155, 267, 473
syntax, 15, 25, 47, 48, 63, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137
Lexical-, 13, 50, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73
Semantic-, 18
Syntax-, 15, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137
ANTLR, 75, 93, 128

Architecture
Embedded-, 137
Intel-, 391
Arguments
Actual-, 223
Call-, 5, 187–190, 192, 193, 197, 206–208, 258, 259, 285, 287, 288, 372, 442, 443
Five-, 218
Formal-, 133, 174, 181, 257, 371
Function-, 208, 248
Input-, 337
Line-, 184, 223, 224, 375
Macro-, 442, 443
Reference-, 207
Relating-, 328
Request-, 383
Separated-, 339
Substitute-, 444
Arithmetic
-Expression, 368
-Manipulations, 320
Array
-Address, 287
-References, 280
-Return, 187
Additional-, 211
Address-, 42
Arguments-, 337
Associative-, 337
Bit-, 204
Cell-, 58
Character-, 439
Code-, 38
Complete-, 324
Data-, 43
Dimensional-, 279, 280
Dimensioned-, 221
Dynamic-, 186, 475
End-, 211, 212
Handle-, 320
Handling-, 287
Implement-, 47
Linear-, 241
Link-, 171
List-, 443
Local-, 280
Over-, 281
Pass-, 280
Pipe-, 383
Pointer-, 215, 315
Required-, 221
Sized-, 280
Sorted-, 439
Struct-, 168, 172
Symbol-, 453
Term-, 287
Time-, 324
Use-, 280, 281
Variables-, 187
Arrays
-Return, 187
Variables-, 187
Assembler
-Directives, 357, 358
-NASM, 275
-Optimization, 13
Generation-, 13
GNU-, 216, 217, 225, 275, 391
Level-, 275
Line-, 434, 436
Macro-, 7, 436, 441–443
Pass-, 254, 436, 437
Standard-, 134
System-, 268, 270, 350
Assembly
Actual-, 436
Conditional-, 434
Define-, 287
Example-, 436
Experienced-, 267
Generated-, 384
Generates-, 306
Get-, 224
Good-, 289
Input-, 435
Intel-, 217, 390
Line-, 275
Macro-, 3, 4
Nonoptimized-, 350
Optimized-, 350, 351
Output-, 382
Own-, 360
Quality-, 300
Raw-, 367–371, 384
Represent-, 275
Sample-, 223
Simplified-, 270
Small-, 435
Translates-, 308
-Code, 302, 303, 306
-Conversion, 24
-Language, 3, 13, 22, 223, 265, 275, 348, 374, 377, 434, 435
Assembly
-Languages, 434
-Program, 261
-Syntax, 390
Assignment, 4, 166, 250, 368, 378
Assignments, 250, 378
Associativity, 63, 91, 103, 136
AST, 73, 162, 163, 176, 227, 239, 240, 260, 265, 270, 323, 350, 370, 380, 384, 385, 387
Atoms, 4, 9–11, 24, 25, 31, 48, 50–52, 75, 384
Attribute
-Acquired, 165
-Env, 142
-Get, 158
-Grammar, 139, 142, 153, 154
-Grammars, 153
Attributes, 58, 134, 135, 139, 151, 153–160, 164, 165, 180, 209, 257, 439, 473, 474
Attributes
Inheritance-, 165
Inherited-, 139, 158
Synthesized-, 139, 155
Auto
-Memory, 334
Automata-, 2
Bounded-, 415
Down-, 415, 424, 427
Finite-, 415, 418
GNU-, 445
State-, 53

Automata
Bounded-, 415
Down-, 415, 424, 427
Finite-, 415
State-, 53
Automaton
Finite-, 418

B

Backus
-Normal, 412, 414
Bacus Naur, 27, 47, 266
BASIC, 6, 22, 81, 188, 224, 475
Basic
-Action, 4
-Architecture, 249
-Assignment, 250
-Backus, 47
-Block, 136, 310, 311, 314, 315
-Blocks, 311, 314, 315
-CPU, 182, 221
-Cannot, 436
-Characteristics, 186, 291
-Collection, 309
-Computing, 355
-Concepts, 347
-Data, 338
-Environment, 183, 184
-Expressions, 473
-Functions, 50
-Grammar, 137, 384
-Groups, 206
-Hardware, 183
-Idea, 26, 35, 230, 440
-Ideas, 26
-Instructions, 347
-Language, 73, 221
-Mathematical, 392
-Mathematics, 221
-Memory, 248
-Method, 191
-Objects, 22
-Operations, 267, 419, 420
-Parser, 138
-Parsing, 75
-Problem, 330
-RDP, 232, 240
-Registers, 248
-Requirement, 269
-Requirements, 269
-Running, 285
-Scanning, 51
-Semantic, 150
-Set, 402, 417
-Sets, 402
-Steps, 70
-Structure, 224
-Synchronizing, 308
-Timer, 222
-Tree, 363
-Tricks, 458
-Types, 167
-Unit, 282
-Virus, 333
BBs, 311, 314, 315
Bennett, 165, 331
Binding
Bindings-, 142
Binutils, 225
Bison, 68, 73, 127, 128, 180
Block
Active-, 211
Allocated-, 204
Auxiliary-, 210
Basic-, 136, 269, 310–312, 314, 315, 317, 320, 325, 326, 331
Buddy-, 204
Building-, 368
Calling-, 242
Code-, 264, 337
Containing-, 337
Contiguous-, 202
Control-, 216
Current-, 199, 214, 215
Enclosing-, 214
Free-, 203, 204
Fully-, 198, 199
Global-, 197
Inner-, 214
Largersized-, 204
Length-, 203
Memory-, 380
Minimum-, 204
Nested-, 209
Next-, 203
Outer-, 209, 224
Partially-, 197, 209
Particular-, 210, 211
Program-, 314
Sized-, 203, 204
Super-, 219
Unmarked-, 204
Unused-, 204
Used-, 204
-Detection, 314
-Form, 312
-Found, 310
-Nesting, 197
-Ptr, 211, 212
-Structure, 209
Blocks
-Form, 312
-Found, 310
Allocated-, 204
Basic-, 269, 310–312, 314, 315, 326, 331
Code-, 264, 337
Free-, 203, 204
Global-, 197
Inner-, 214
Length-, 203
Memory-, 380
Program-, 314
Sized-, 203, 204
Unmarked-, 204
Unused-, 204
BNF, 22, 27, 47, 79, 80, 113, 114, 128, 154, 266, 414, 463
Branch
-Code, 377, 378
-Label, 143, 253
-Next, 143, 144
-Operations, 144
Bytecode, 1, 4, 10, 227, 244

Bytecode
-Interpreter, 9, 185
-List, 243
-Program, 243
-Represents, 4
-Used, 241

C

Calculator, 120, 126, 127, 137, 140, 151, 163, 233, 307, 348, 349
Call, 66, 187, 189, 207, 217, 242, 246, 247, 265, 284, 378, 475, 476, 478
Calling
-Block, 242
-Code, 206
-Convections, 206
-Convention, 188–191, 206, 224, 225, 257, 283, 284, 286, 288, 325, 375, 384
-Conventions, 206, 224, 225, 257, 283, 286, 288, 375, 384
-Func, 150, 189, 206, 285
-Function, 189, 206, 285
-Functions, 285
-Idioms, 288
-Main, 193
-Mechanism, 286
-Program, 266, 267
-Routine, 196, 202, 284
Calls
-Eax, 383
-Fibonacci, 244
-Int, 183, 218
-Interface, 183, 218
-Issued, 220
-Yylex, 63
Function-, 4, 176, 188, 221, 248, 249, 258, 261, 283, 357, 451
Interpreter-, 176
Macro-, 443
Nested-, 283
Output-, 287
Recursive-, 283
Repeated-, 283
Repeatedly-, 115
Routine-, 188, 285
Subroutine-, 188
System-, 182, 183, 216–220, 287, 357, 374, 382, 383, 451
Canonical, 77, 103
Cantor, 407, 408
Cardinality, 400
CFG, 50, 72, 73, 75, 78, 79, 81, 85, 87, 91, 92, 154, 331, 424–426, 429, 430, 460, 462, 463, 473
Chomsky
-Classification, 415
-Classified, 415
-Level, 73
-Normal, 426
-Type, 50, 76, 414, 415
CISC, 188, 218, 270, 271
Classification
Chomsky-, 415
Closure, 104, 402, 411
Code
binary, 3
machine, 2, 9–11, 22, 30, 38, 41, 43, 44, 46, 63, 228, 229, 241, 245, 246, 300, 334, 341, 346, 435, 448, 450
operation, 3, 7, 29, 30, 217, 243, 245, 247, 266, 275, 303, 309, 312, 434–438
-Dead, 322
-Elimination, 322
-Gen, 11, 13, 18, 41, 51, 62, 185, 228, 268, 269, 271, 273, 275, 277, 279, 281, 283, 285, 287, 289, 291, 293, 295, 297, 299, 301, 303, 305, 307, 309, 367
-Generation, 11–13, 18, 41, 185, 268, 269, 271, 273, 275, 277, 279, 281–283, 285, 287, 289, 291, 293, 295, 297, 299, 301, 303, 305, 307, 309, 367, 384
-Generator, 62, 266, 269
-Interactively, 185
-Interpreter, 9, 185
-Library, 185
-Loaded, 8
-Motion, 324
-Optimization, 8, 18, 238, 289, 310, 311, 313, 315, 317, 319, 321, 323, 325, 327, 329, 331
-Optimizer, 261
-Parser, 269
-Program, 266
-Public, 265
-Represents, 4
-Target, 269
-Void, 231
Analysis-, 13
Assembly-, 302, 303, 306
Branch-, 377, 378
Compiled-, 44
Compiler-, 9, 185
Dead-, 322
End-, 11
Independent-, 457
Intermediate-, 62, 227–229, 231, 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, 253, 255, 257, 259–261, 263, 265, 267
Code
Machine-, 8, 30, 185, 241
Mapper-, 13
Optimized-, 19
Scanner-, 269
Source-, 8, 51, 185
Threaded-, 244–247
Tuple-, 239
Coded
Explicitly-, 65
Codes
ASCII-, 61
Byte-, 4
Condition-, 357
Contain-, 4
Driver-, 20
Error-, 382, 437
Intermediate-, 227, 229, 261

Invocation-, 288
Readable-, 264
Source-, 30, 440
Threaded-, 227
Coercion, 166, 167, 174, 175, 180
Collection
Aggregation-, 225
Compiler-, 134, 137, 199, 265, 328
Collector
Garbage-, 202, 205
Command
Actual-, 223
Appropriate-, 334
Binary-, 2
Commands-, 1, 24, 25, 63
Compilation-, 220
Console-, 183
Elementary-, 2
Executes-, 24
Float-, 387
Giving-, 117
Interpretation-, 140
Invocation-, 183, 223
Jump-, 311
LABI-, 256
Main-, 345
Option-, 222
Own-, 2
Proper-, 375
Run-, 386, 387
Shell-, 445
Specified-, 220
Specifies-, 4
Specifying-, 69, 124
Stepping-, 387
Commands
Elementary-, 2
Executes-, 24
Interpretation-, 140
Jump-, 311
Option-, 222
Shell-, 445
Specifies-, 4
Comments, 42, 44, 49, 53, 55, 66, 90, 154, 291, 435, 445
Common
-Among, 209
-Calculations, 281
-Environment, 222
-Evaluation, 207
-Exception, 312
-Imperative, 4
-Intel, 341
-Language, 341
-Operation, 369, 442
-Point, 430
-Practice, 325
-Programming, 339
-Rules, 209
-Sequences, 308, 328
-Set, 398
-Sub, 79, 323, 324
-System, 216
-Terms, 431
-Use, 348
-Used, 4, 227
Commonly-, 4, 227
Elements-, 432
Most-, 207, 209, 279, 339, 348
Move-, 281
Quite-, 209, 221
Compile
-Back, 12
-ByteCode, 9, 185
-Code, 44
-Collection, 134, 137, 199, 265, 328
-Compiler, 114
-Construction, 24, 128, 181, 225
-Data, 185
-Front, 12
-Generator, 137
-Language, 11
-Source, 11, 228, 449
-Writing, 61, 75, 93, 114
Compiler
About-, 27
Best-, 19
COBOL-, 244
Code-, 338, 346
Coded-, 62
Compact-, 332
Complete-, 21, 384
Debugging-, 228
Define-, 447
Detailed-, 137
Develop-, 12
Developing-, 46
Emphasized-, 22
Example-, 58
Faster-, 9
First-, 92
FORTRAN-, 9
Free-, 73
Full-, 114, 133, 134
GCC-, 204, 226, 260
Gcc-, 275, 330
GNU-, 73, 134, 135, 137, 199, 260, 261, 265, 266, 328
Good-, 5, 182, 314
Gradually-, 21
His-, 330
HLL-, 5, 9 434
Interactively-, 185
Java-, 241, 264, 333, 334
Language-, 167, 221, 341, 424
Legacy-, 191
McCat-, 260
MiniC-, 286, 288, 348, 350, 362, 364
Modern-, 7, 10, 11
Most-, 329
New-, 113
Optimizing-, 10, 134, 228, 269, 289, 328, 346
Ordinary-, 346
PASCAL-, 241
Pass-, 9
Powerful-, 62
Program-, 11
Purdue-, 128
Real-, 10, 292, 322
Simpler-, 8
Simplify-, 62
Theoretical-, 20
Time-, 10

Compiler (continued)
Typical-, 434
Use-, 46
World-, 160
-Back, 12
-ByteCode, 9, 185
-Collection, 134, 137, 199, 265, 328
-Construction, 24, 128, 181, 225
-Data, 185
-Front, 12
-Generator, 137
-Language, 11
-Source, 11, 228, 449
-Writing, 61, 75, 93, 114
Compilers
COBOL-, 244
Develop-, 12
Developing-, 46
Good-, 5
Java-, 241
Language-, 341
Modern-, 10, 11
Optimizing-, 10
PASCAL-, 241
Pass-, 9
Real-, 292
World-, 160
Compiling , 165, 293, 331
Component
Components-, 7
Connected-, 226
Functional-, 73
Important-, 115
Major-, 115
Processor-, 269
Program-, 171, 172
Respective-, 120
Single-, 168
Standard-, 336
Value-, 52
Components
Connected-, 226
Functional-, 73
Important-, 115
Major-, 115
Processor-, 269
Program-, 171, 172
Computation, 280, 392, 419
Computing
-Surveys, 308
ACM-, 308
Conditional
-Directive, 447
-Jump, 184, 310, 357
-Jumps, 310
Conflict
-Alternate, 137
-Cannot, 106
-Occurs, 138
-Resolution, 103, 110, 121, 138, 384
-Resolutions, 384
-Situations, 107
-State, 108
Finds-, 121
Reduce-, 102–105, 108, 109, 121–123, 127, 136, 138, 468, 470, 471
Removing-, 136
Resolve-, 127
Unresolved-, 127
Conflicts
-Alternate, 137
Finds-, 121
Reduce-, 102, 123, 127
Removing-, 136
Resolve-, 127
Unresolved-, 127
Constant
-Folding, 317, 321
-Load, 277
-Propagation, 321
-Value, 52
Constants-, 52
Entry-, 52
EXPR-, 261
Load-, 308
Reals-, 242
Table-, 52
Constructors, 168, 360
Constructs
Additional-, 414
Aggregating-, 384
Code-, 278
Complex-, 159
Control-, 5, 50, 75, 102, 141, 149, 164, 234, 249, 252, 268, 310, 337, 368, 384
Doubtful-, 167
Expr-, 179
Jump-, 299
Language-, 25, 27, 51, 173, 475
Longer-, 445
Looping-, 314
Lower-, 153
Nested-, 237
Original-, 240
Process-, 64
Program-, 167, 249
Programming-, 289
Sensitive-, 141
Statement-, 134, 141, 435
Syntactic-, 11, 15
Unacceptable-, 28
Context
-Free, 412
-Information, 298
-Languages, 412
-Parameters, 300
-Rules, 50, 76
-Sensitive, 50, 76
-Symbols, 138
Enumerable-, 415
Extra-, 299
List-, 337
Right-, 104
Scalar-, 337
Specified-, 298
Use-, 50
Valid-, 160
Control
-Constructs, 282
-Flow, 346
-FSM, 145, 427
-Push, 79
-Word, 388

Symbol-, 145
Typing-, 216
Conventions
Call-, 350
Calling-, 206, 224, 225, 257, 283, 286, 288, 375, 384
Current-, 193
Naming-, 63
Conversion
Assembly-, 306
Explicit-, 335
Show-, 335
Tuple-, 306
Type-, 166, 173–175, 334, 335
Converting, 302
CPU
-ALU, 184
-Architecture, 113, 188
-Architectures, 113, 188
-Cache, 276
-Differing, 270
-Families, 19
-Family, 3, 434
-Hardware, 79, 184, 221
-Instruction, 113
-Plus, 184
-Provide, 274
-Register, 18, 113, 187, 188, 216, 218, 238, 269–272, 274, 287, 291, 387
-Registers, 18, 113, 187, 188, 216, 218, 238, 269–271, 274, 291
-Starts, 290
-System, 183
-Time, 216, 269
-Tries, 290
-Type, 272
Basic-, 182, 221
Call-, 184
Day-, 184
Hardware-, 260
Implement-, 434
Includes-, 183
Integer-, 308
Modern-, 79, 192, 245, 275
Modified-, 113
New-, 113
Particular-, 23
Required-, 188
Saved-, 216
Special-, 188
Traditional-, 22
Used-, 274
Cross
-Library, 451
-Source, 208
Cycle
-Directed, 432
-Stealing, 308
Graph-, 432

D

DAG
-Directed, 47
Dangling
-Else, 102, 103
Dead
-Code, 19, 322, 323
-Var, 315, 322, 323
-Variable, 322, 323, 331
Debugging, 4, 5, 19, 25, 69, 114, 116, 117, 124, 134, 147, 211, 220, 228, 287, 288, 346–348, 384, 388, 396, 406, 475
Declarations
Variable-, 174, 257
Definition, 4, 77, 85, 87, 90, 96, 105, 139, 154, 155, 157, 192, 196, 257, 309, 371, 394, 395, 403, 410, 411, 417, 419, 425–428, 443, 453, 471
Definitions, 139, 154, 155, 157, 192, 196, 257, 403
Dependency
-Graph, 139, 155, 157
-Graphs, 155, 157
Associated-, 155
Circular-, 157, 158
Dependent
Functions-, 308
Highly-, 220
Less-, 220
Machine-, 5, 11, 37, 228, 261, 273, 302, 320
Platform-, 222, 333
Position-, 448, 457
Substantially-, 289
Techniques-, 48
Tested-, 375
Derivation
Attribute-, 156
Canonical-, 425
Later-, 410
Leftmost-, 77–79, 87, 426, 464
Possible-, 426
Resultant-, 425
Right-, 78
Rightmost-, 78, 89, 92, 93, 104, 426, 464
Sentence-, 154
Derivations
Later-, 410
Leftmost-, 426
Right-, 78
Rightmost-, 92, 93, 464
Deterministic
-FSM, 137, 418–421
-Finite, 72
-PDA, 427
About-, 420
Parsed-, 78
DFA
-Deterministic, 73
-Model, 364
DFAs-, 72
DFSM
-Recognizing, 423
Converting-, 422
Equivalent-, 421
Example-, 420, 421
Small-, 420
Diagram
-Output, 7
-Sequence, 423
Railroad-, 414
State-, 108, 109, 115, 116, 118, 119, 137, 419, 431
Transition-, 94, 418, 423

Diagrams
Railroad-, 414
State-, 108
Digraph
-Representation, 401
Digraphs-, 401
Direct
-Acyclic, 47
-Address, 30, 309, 381
-Addressing, 381
-Derivation, 77
-Derivative, 413
-Graph, 401, 432
-Proof, 404
-Threaded, 245, 247, 266
Assembler-, 357
Canonical-, 77
Conditional-, 447
Cycle-, 432
DAG-, 47
Linkage-, 450
Directed
-Graph, 401, 432
Cycle-, 432
Directive
Conditional-, 447
Directives-, 358
Endr-, 358
Equ-, 380
Driven
Completely-, 154
Slice-, 204
Dynamic
-Allocation, 190
-Library, 452
-Link, 198, 452
-Linking, 451, 452
-Memory, 188
-Semantics, 142
-Storage, 186, 475
Locals-, 198

E

Effective
-Address, 309
Embedded
-Actions, 153
Empty
-Alphabet, 408
-Cell, 146
-List, 464
-Page, 459
-Parameter, 241
-Path, 314
-Production, 112, 464
-Productions, 112
-Rules, 136
-Set, 403, 409, 410
-Stack, 428
-Statement, 121
-String, 85, 409, 410, 418
Entities
Created-, 337
Defined-, 4, 131
Specified-, 4
Syntactic-, 34
Entry
Correct-, 254
Dictionary-, 344
Main-, 66
Matching-, 340
MNT-, 444
Next-, 38
Operation-, 39
Program-, 216, 326, 449
Remote-, 253
SymTab-, 318
Table-, 52, 58, 152, 163, 186, 215, 257, 439
Text-, 217
Tuple-, 249, 252–255, 312
Environment
-Variables, 183, 222
Basic-, 183, 184
Common-, 222
Current-, 173
Development-, 69, 128, 134, 334, 341
Indicates-, 174
Individual-, 184
Linux-, 455
Memory-, 182
New-, 142
Operating-, 221, 287
Overall-, 183
Programming-, 127, 176, 348, 349
Runtime-, 182
System-, 182, 183, 222
Tasking-, 185
Time-, 6, 21, 47, 182, 183, 185–189, 191, 193, 195, 197, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 221, 223–225, 269, 287, 333, 334, 475
Total-, 183, 184
Environmental
-Factors, 185
-Parameters, 182, 183
-Values, 183
Equ, 310, 435, 437
Equivalence
-Notations, 395
-Relation, 432
-Relations, 402
String-, 432
Error
-Checking, 11, 13
-Handling, 70, 123
-Table, 437
-Type, 114
ASD-, 71
ESD, 453, 454
Evaluation
-Sequence, 275
Examples
-Gopal, 395
Easy-, 359
Further-, 143
Good-, 241
Indicative-, 368
Matching-, 64, 65
Working-, 125
Exception, 309
Execution, 2, 4, 6, 8–11, 23, 25, 39, 44, 142, 143, 184–186,

188, 198, 202, 206, 208, 210, 220, 226–228, 241, 244, 248, 266, 267, 272, 275, 289, 291, 300, 301, 309, 314, 315, 320, 331, 336, 342–344, 346, 359, 387, 396, 406, 439, 447, 448, 451, 455, 475–478
Exercise, 72, 163–165, 172, 224, 264, 349, 384, 431
Exercises, 163, 264, 349, 431
Expression
-Abc, 65
-Analysis, 326
-Conforming, 73
Arithmetic-, 368
DECL-, 261
Examples-, 65
Regular-, 50, 73, 392, 412, 414, 417
Type-, 168
Expressions, 13, 15, 24, 50, 51, 53, 63, 69, 72, 75, 79, 103, 137, 151, 161, 164, 173, 177, 206, 207, 232–234, 248, 249, 266, 292, 294, 300, 314, 317, 323, 326, 328, 331, 337, 348, 395, 407, 412, 417, 424, 425, 431, 447, 473, 474
Expressions
Regular-, 50, 392, 412, 414, 417
Type-, 168
Extended, 3, 26, 39, 79, 80, 86, 129, 146, 192, 234, 250, 309, 325, 360, 414, 445, 463
Extension, 4, 8, 47, 64, 114, 127, 128, 193, 199, 221, 229, 296, 328, 349, 355

F

Facilities, 4, 5, 19, 48, 69, 119, 124, 128, 131, 150, 151, 184, 248, 266, 291, 332, 348, 349, 424, 441, 444, 447
Falls, 290, 295
Fetching, 301, 338
Files, 8, 13, 19, 69, 114, 126, 127, 133, 183, 187, 208, 219–222, 224, 270, 287, 315, 334, 337, 338, 350, 358, 368, 370, 375, 383, 385, 399, 445–447, 455, 456
Finding
-FOLLOW, 87
-Parameters, 283
Finite
-Automaton, 418
FIRST , 85–88, 104, 380, 460–463
Flex, 21, 30, 46, 50, 62, 63, 68, 69, 72, 73, 114, 117, 142, 146, 231, 270, 286, 297, 364, 401–403, 424, 432
Floating
-Point, 184, 271, 308
Flow, 49, 136, 196, 213, 244, 260, 267, 290, 310, 312, 314, 315, 325–327, 329, 331, 346, 354–357, 359
Flow
-Control, 357
Example-, 313
Folding
Constant-, 321
Variable-, 324
FOLLOW, 86–88, 100, 104, 106–112, 460–463, 468, 470, 471
Formal
-Algorithm, 311
-Arguments, 174
-Grammars, 413
-Lang, 2, 140, 392, 393, 395, 397, 399, 401, 403, 405, 407–409, 411–413, 415, 417, 419, 421, 423, 425, 427, 429, 431, 433
-Language, 2, 392, 393, 395, 397, 399, 401, 403, 405, 407–409, 411–413, 415, 417, 419, 421, 423, 425, 427, 429, 431, 433
-Languages, 2, 392, 393, 395, 397, 399, 401, 403, 405, 407–409, 411–413, 415, 417, 419, 421, 423, 425, 427, 429, 431, 433
-Logic, 392
-Semantics, 173
Formalism, 166
Forms, 1, 7, 22, 37, 77, 78, 88, 89, 123, 227–229, 262, 266, 277, 278, 308, 309, 315, 326, 332, 334, 345, 346, 355, 360, 364, 398, 410, 422, 426, 427, 435, 465
FORTH, 208, 227, 228, 233, 244, 332, 341, 342, 346
Forth
-Advocates, 341
-Definitions, 346
-Implementation, 341
-Implementations, 341
-Jargon, 342
-Program, 341, 345
-Programming, 341
-Programs, 345
-Standard, 341
-Systems, 346
-Uses, 345
-Word, 343, 345
ANS-, 341, 344
Intel-, 341
ISO-, 341
Frames, 241, 458, 477
Framework, 47, 261, 269, 316
Free
Area-, 187
Data-, 451
Freeware-, 24
Name-, 415
Stack-, 191
Front, 12, 73, 84, 85, 114, 128, 134, 177, 182, 226, 260, 266, 291, 315, 329, 336, 346, 347, 350, 351, 412, 431, 445
Front
-End, 73, 265
-Ends, 73
FSM, 13, 15, 50, 53, 55, 56, 63, 64, 72, 75, 79, 81, 82, 93–95, 97, 100, 105, 110, 115, 118, 127, 137, 138, 145, 415, 417–424, 427, 430, 431, 470
FSMs, 50, 53, 418–421

Function
-Arguments, 208, 248
-Assumed, 448
-Bindtype, 178
-Body, 174, 207, 208, 265
-Calculator, 120
-Call, 4, 66, 80, 176, 177, 188, 189, 191, 193, 202, 206, 207, 221, 224, 248, 249, 258, 261, 283, 284, 286, 288, 322, 324, 325, 350, 357, 371, 372, 378, 384, 406, 431, 451
-Calling, 191, 224, 286, 384
-Calls, 4, 176, 188, 221, 248, 249, 258, 261, 283, 357, 451
-Compiled, 288
-Components, 73
-Connecting, 432
-Consists, 181
-Dealing, 145
-Declaration, 177
-Declared, 209
-Decreases, 285
-Defined, 133, 262
-Definition, 176, 196, 209, 249, 251, 257, 288, 309, 371
-Definitions, 176, 196, 249, 257
-Described, 338
-Determine, 283
-Eargpush, 372
-Efadd, 368
-Encounters, 208
-Ending, 381
-Epilogue, 285, 371
-Error, 176
-Etc, 131
-Eval, 176, 177
-Evals, 177
-Factorial, 194
-Get, 54, 177, 184, 203, 223, 315
-Getenv, 184
-Getmat, 315
-Getopt, 223
-Gettypeid, 177
-Graph, 432
-Header, 222
-Ignores, 212
-Inadvertently, 224
-Included, 133
-Increments, 235
-Independent, 308
-Inserts, 240
-Install, 33
-Int, 126
-Integer, 126
-Invocation, 284, 372
-Language, 176, 180, 248, 332
-Languages, 248
-Let, 400
-Libraries, 133
-Library, 8, 184, 375
-Liveness, 315
-Lookup, 33
-Lprec, 90
-Main, 192, 194, 262, 263
-Might, 286
-Mkbb, 315
-Mksymtab, 315
-Must, 283
-Myfunc, 286, 305
-Name, 66, 176, 208, 283, 364, 371
Function
-Names, 208, 364
-Need, 36, 131
-Needed, 131
-Needs, 285
-Number, 218
-Onto, 378
-Parameters, 206, 225, 283, 286
-Parg, 371, 372
-Pargassgn, 372
-Pfassgn, 368
-Pointers, 202
-Pop, 148
-Printf, 206
-Printing, 230
-Programming, 5, 22, 172
-Prologue, 285, 371
-Properly, 66
-Pus, 117
-Rec, 240
-Recognizes, 240
-Reference, 208
-Retrieves, 193
-Return, 172, 193, 206, 207, 225, 251, 258, 283, 285, 337
-Returns, 172, 193, 207, 283, 285
Function
-Saves, 285
-Setupbranch, 315
-Storage, 478
-Strlen, 375
-Surjection, 400
-Switches, 372
-Terminates, 285
-Test, 189, 190, 288
-Traverse, 264
-Type, 166
-Use, 363, 370, 372
-Used, 260, 370
-Written, 288, 350
-Yylex, 63, 69, 115
-Yyparse, 63, 115
-Yywrap, 66
Acceptance-, 10
Access-, 348
Action-, 100
Additional-, 131
ADT-, 363, 384
Allocation-, 434
Analysis-, 349
Analyzer-, 115
Associated-, 202
Auxiliary-, 165
Based-, 180, 390
Basic-, 50
Call-, 284
Calling-, 189, 206, 285
Compact-, 337
Compiler-, 191, 224
Consuming-, 19
Contains-, 351
Current-, 241

Database-, 448
Debug-, 69
Declarations-, 249
Defined-, 128, 133, 209, 371, 373, 476, 477
Function
Defining-, 69
Delete-, 364
Delimiter-, 147
Dump-, 147
ECHO-, 66
Enclosing-, 202, 226
Essential-, 231
Evaluation-, 208
Evaluator-, 176
First-, 115, 208
Generation-, 369
Generator-, 369, 371, 372
Goal-, 328
GoTo-, 100
Handle-, 176
Handles-, 176
Handling-, 59, 147, 148, 176, 448
Hash-, 317
Hashing-, 331, 440
Includes-, 63
Indirect-, 357
Initializer-, 364
Insert-, 254
Involve-, 477
Library-, 184, 203, 204, 206, 221, 223, 349, 451
Line-, 325
LOOKBACK-, 112
Lookup-, 212
Main-, 193, 336
Major-, 46
Management-, 146, 240, 459
Manipulation-, 59, 180, 249, 315, 337
Map-, 5
Mapping-, 421
Mathematical-, 361
Nested-, 196, 199, 201, 202, 226, 283, 309
Numerical-, 271
Operand-, 329
Own-, 341, 348
Parse-, 468
Parser-, 115, 117
Parsing-, 81, 82, 85, 86, 88, 135, 240, 461, 462, 468
Function
Partial-, 400
Pass-, 283
Pop-, 146
Primitive-, 244
Printr-, 390
Processing-, 399, 442
Provide-, 448
Provides-, 183
RDP-, 236, 237
Recursive-, 39, 145, 185, 188, 194, 283, 406
Relation-, 432
Required-, 350
Results-, 8
Scanner-, 170
Search-, 60
Separate-, 54
Sine-, 174
Small-, 176
Special-, 184, 221, 271, 343
Stack-, 249
Standard-, 381
Table-, 391
Terminal-, 148, 149, 235
Total-, 400
Transition-, 418, 419, 421, 427, 428
Traversal-, 232
Tree-, 221
Typical-, 188, 232
Undefined-, 209
Unusual-, 332
Useful-, 337
Userdefined-, 128
Utility-, 149, 180, 260, 351
Valid-, 371
Value-, 329
Functions, 19, 22, 36, 39, 46, 50, 58–60, 66, 69, 86, 90, 100, 101, 112, 115, 131–133, 141, 146–149, 165–167, 170, 176, 178, 180, 183–186, 188, 191, 202, 203, 206, 208, 209, 220, 221, 226, 231, 235, 237, 240, 244, 249, 261, 262, 264, 268, 271, 283, 285, 288, 308, 309, 315, 328, 329, 337, 338, 341, 348–351, 361–363, 369–372, 375, 384, 386, 390, 391, 399, 400, 424, 434, 442, 445, 448, 468, 476, 477
Functions
-Let, 400
-Library, 375
-Surjection, 400
-Used, 260
Nested-, 199
Recursive-, 406
Total-, 400
Utility-, 260

G

Garbage, 202, 204, 205, 225, 266, 334
GCC, 19, 20, 73, 134, 135, 137, 199, 202, 204, 226, 227, 260–262, 265, 266, 275, 287, 288, 328, 382
Gcc, 19, 20, 23, 63, 73, 126, 127, 137, 193, 200, 206, 222, 224, 225, 260, 261, 264, 265, 269, 275, 286, 311, 330, 350, 382, 447, 452, 455
Genealogy , 6
Generating
-Abstract, 240
-RPN, 235
Generation, 11, 12, 21, 22, 26, 39, 41, 42, 47, 62, 63, 85, 100, 114, 134, 144, 160, 182, 185, 230, 234, 236–240, 249, 251, 252, 254, 255, 257, 258, 260, 265, 268, 269, 271, 273–275, 277, 278, 282, 287, 289, 291, 303, 308, 324, 329, 333, 346, 348, 349, 367, 369, 384, 391

Generator
-Coco, 137
-Object, 62
Analyzer-, 68, 73, 114
Code-, 30, 42, 62, 253, 262, 266, 269, 270, 278, 286, 291, 302, 303, 306–308, 310, 369, 371, 372, 384, 385, 424
Compiler-, 123, 137
Independent-, 261
Parser-, 46, 68, 73, 80, 112, 113, 127, 128, 150
RPN-, 237
RTL-, 261
Scanner-, 30, 46
Target-, 269
Global
-Data, 310, 325
-External, 453
-Optimization, 325
-Variable, 199, 378
-Variables, 378
Assignments-, 378
GEST-, 453
GLR, 127, 138
GNU, 12, 19, 68, 73, 114, 127, 134, 135, 137, 180, 193, 199, 204, 206, 216, 217, 225, 227, 260, 261, 265, 266, 269, 275, 287, 311, 328, 330, 331, 341, 353, 384, 386, 390, 391, 445, 455
Grammar
LL, 92
LL(1), 75, 81–83, 85–87, 460–463
LL(k), 78
LR, 92, 112, 128
LR(0), 94, 100, 102, 105, 106
LR(1), 102–104, 106, 112, 467, 468, 470, 471
LR(k), 89
Grammar
Acceptor-, 412
Ambiguous-, 76, 91, 113, 136, 425, 426
ANTLR-, 128
Attribute-, 21, 139, 142, 153–157, 163, 474
Attributed-, 384
Augmented-, 151, 152
Bare-, 152
Basic-, 137, 384
BEC-, 137
Bison-, 127
BNF-, 154
Candidate-, 460
Complex-, 79
Conventional-, 154
Distinct-, 136
Earlier-, 121
Enhance-, 47
Enhanced-, 47
Equivalent-, 431
Example-, 15, 81, 85, 87, 91, 94, 97, 101, 104, 107, 112, 145, 146, 229, 235, 415
Expression-, 77, 80, 81, 90, 156, 239
Extensive-, 145
Final-, 161
Formal-, 413
Free-, 47, 73, 89, 112, 145, 424
Full-, 51, 161
GIMPLE-, 260
Indian-, 412
Input-, 128
Integer-, 155
Language-, 22, 30, 48, 113
Less-, 426
MiniC-, 255, 348, 351
Modified-, 123
Operator-, 90
Original-, 116
Parse-, 154
Partial-, 152
Particular-, 93, 109, 138
PDA-, 415
Precedence-, 90
Recursive-, 79
Reduced-, 78
Regular-, 61, 63, 73, 417, 422
Restricted-, 145
Sample-, 240
Sanskrit-, 412
Scanner-, 31
Sensitive-, 50, 75, 141
SLR-, 112
Systematic-, 412
Unambiguous-, 91, 113
Useful-, 93
Uses-, 63
Valid-, 138
Yacc-, 62, 128, 160–162, 164, 249, 384
Grammars
Ambiguous-, 113
Attribute-, 21, 139, 142, 153, 154
Bison-, 127
Distinct-, 136
Enhanced-, 47
Formal-, 413
Free-, 47, 89, 112, 145, 424
Sensitive-, 50, 75
SLR-, 112
Unambiguous-, 113
Graph, 22, 37, 47, 127, 139, 155, 157, 164, 165, 172, 184, 221, 226, 267, 310, 312–314, 326, 327, 331, 347, 401, 432, 433, 476–478
Graph
-Intersection, 432
-Representation, 401
Call-, 475
Dependency-, 155, 157
Function-, 432
Graphs, 155, 157, 226, 310, 314, 401
Graphs
Dependency-, 155, 157
Greibach, 426
Growth, 347
GSO, 328

H

Handling, 44, 55, 58, 59, 72, 81, 114, 123, 128, 139, 147, 148, 176, 189, 237, 287, 334, 364, 424, 440, 444, 448

Handling
Error-, 70, 123
Table-, 159, 439
Hash, 160, 215, 280, 315, 317, 331, 337, 338, 349, 398, 399, 440
Hash
-Data, 440
-Identifier, 215
-Table, 58, 215, 440
Based-, 215
Data-, 440
Use-, 399
Header, 9, 32, 117, 152, 205, 209, 213, 214, 221–223, 230, 381, 399, 445, 446
Heap
-Allocation, 202
-Data, 266
-Memory, 266
-Objects, 205
-Space, 205
-Store, 242
Heuristics, 312, 316, 329
Hexadecimal, 3, 46
High
-Embedded, 137
-Order, 137
-Register, 296
Hoc, 50, 61, 129, 161, 176, 213, 269, 349
Holub, 137

I

Idea, 5, 12, 19, 21, 24, 26, 35, 62, 74, 90, 91, 113, 114, 145, 174, 204, 215, 226, 230, 235, 244, 245, 261, 283, 308, 310, 325, 329, 335, 338, 349, 368, 392, 408, 419, 430, 440, 441, 458
Identifiers, 7, 27, 35, 48, 50–53, 55, 56, 58, 61, 62, 67, 75, 131, 160, 167, 179, 209–213, 215, 224, 413, 431, 448, 474
Immediate
-Address, 309
Implementation
-Issues, 475
-Notes, 396, 398, 400, 424
Example-, 314
Implemented, 8, 11, 20, 21, 24, 26, 30, 113, 114, 142, 145, 176, 177, 191, 214, 221, 244–248, 268, 282, 284, 291, 300, 314, 336, 338, 346, 359, 369, 378, 379, 384, 434, 442, 444, 475
INCLUDES, 111, 112
Increment, 42, 201, 205, 206, 235, 247–249, 300, 308, 321, 324, 355, 369, 377, 379, 438, 443, 444
Independent, 5, 11, 19, 37, 48, 51, 218, 222, 228, 244, 253, 261, 266–268, 273, 289, 291, 302, 308, 310, 311, 315, 320, 333, 334, 375, 443, 448, 449
Index, 30, 42, 48, 52, 62, 63, 73, 120, 152, 158, 165, 180, 215, 221, 234, 236, 237, 279, 284, 287, 296, 309, 314, 317, 324, 336, 431, 439, 440, 444, 455
Indexed, 30, 42, 284, 317
Indexed
-Address, 48, 309
-Addressing, 48
Indirect
-Address, 309
-Threaded, 246, 266
-Threading, 247
-Vector, 280
-Vectors, 280
Individually, 224
Infer, 166, 172, 173, 249, 276, 404
Inference, 166, 172, 173, 249, 404
Inference
-Rule, 404, 405
-Rules, 173, 403, 404
Induction-, 405
Influence, 185, 260, 268, 270, 420
Inheritance, 156
Inherited
-Attributes, 139, 154, 158, 159
Initialize, 3, 47, 141, 163, 191, 255, 322, 351, 358, 364, 380, 388, 450, 451, 455, 459
Install, 33, 56, 57, 59, 132, 164, 180, 213, 264, 336, 362, 365, 366
Instruction
-Commands, 25
-Execution, 184, 300, 301
-Mnemonic, 360
-Pipelining, 300
-Pointer, 29, 188, 296, 388, 457
-Register, 30
-Rescheduling, 300
-Scheduling, 300
-Set, 3, 243, 268, 270, 309, 348, 391
Complex-, 270
Core-, 184
Integer-, 355
Point-, 357
Reduced-, 270
Transfer-, 354
Xffff-, 388
Instructions
Integer-, 355
Point-, 357
Transfer-, 354
Integers, 5, 13, 124, 141, 175–177, 200, 242, 260, 271, 276, 279, 349, 355, 360, 385, 397, 401–403, 406, 409, 431, 474, 475
Interact, 1, 8, 52, 73, 185, 341, 345, 348, 349
Interesting, 2, 22, 127, 193, 253, 320, 332
Interface, 48, 128, 183, 218, 225, 243, 287, 309, 332, 357
Intermediate, 4, 9, 11, 13, 18, 19, 26, 37–39, 41–43, 48, 52, 63, 139, 143–145, 148–150, 152, 161, 187, 227–229, 232, 234, 237, 239, 241, 244, 249, 260–262, 264, 265, 267–269, 277, 284, 289, 292, 293, 333, 348, 349, 367, 370, 371, 443

Intermediate
-Analysis, 11
-Code, 26, 62, 227–229, 231, 233, 235, 237, 239, 241, 243, 245, 247, 249, 251, 253, 255, 257, 259–261, 263, 265, 267
-Codes, 227, 260
-Form, 37
-Language, 10, 227, 228
-Representation, 139, 160, 161, 269, 270, 348
-Rules, 145
-Token, 51
Analyser-, 62
Code-, 43
Common-, 10
End-, 11, 228
GCC-, 261
Input-, 270
Liner-, 51
Optimized-, 269
Internal
-Data, 260, 338
-Fragmentation, 204
-Linkage, 208
-Perl, 338
Collection-, 137, 265
Html-, 260
Perl-, 338
Internalize, 21, 24
Interpret, 1, 2, 7, 8, 11, 21, 22, 25, 29, 46, 51, 73, 90, 103, 115, 140, 142, 152, 164, 170, 175–178, 180, 185, 213, 227, 228, 233, 234, 244, 248, 265, 266, 332–334, 336–338, 340–349, 367, 415, 421
Interpreter, 1, 7–9, 11, 21, 22, 25, 51, 73, 115, 142, 164, 175, 176, 178, 180, 185, 213, 227, 228, 233, 265, 266, 332, 334, 338, 341–347, 349
Interpreters, 1, 233, 266, 346, 347
Intrinsic, 155
Introduction, 22, 128, 227, 232
Introduction
Quick-, 342
Short-, 339
Invariant, 19, 324, 331
Invocation, 138, 181, 182, 187, 188, 192, 196, 198, 223, 235, 243, 244, 258, 265, 267, 284, 287, 288, 372, 442
Invocation
-Command, 183, 223
Involved, 50, 62, 75, 80, 128, 133, 175, 183, 187, 274, 286, 302, 309, 369, 430
Issues, 23, 26, 46, 55, 80, 167, 250, 268, 283, 291, 306, 310, 448, 475
Item, 96–98, 100, 102–107, 109, 110, 112, 118, 119, 138, 153, 193, 207, 224, 246, 247, 266, 309, 343
Iterative, 36, 48, 79, 80

J

Java, 4, 6, 10, 11, 19, 22, 26, 73, 128, 134, 135, 142, 166, 167, 180, 186, 203, 207, 222, 223, 227, 228, 241–244, 264–266, 280, 283, 300, 332–335, 338, 339, 424
Jig, 429
Jump
-Addresses, 450
-Instructions, 44, 310

K

Kernighan, 127, 137, 176, 225, 348, 349
Key, 3, 28, 31, 47, 48, 51–53, 58, 73, 88, 94, 131, 132, 159, 164, 208, 229, 253, 267, 304, 323, 337, 338, 351, 364, 399, 440
Keyword, 28, 31, 47, 48, 51–53, 58, 73, 131, 132, 159, 164, 208, 229, 253, 267, 323, 351, 364

L

LA, 5, 6, 29, 32, 38, 41–43, 75, 93, 107–113, 127, 128, 145, 150, 175, 222, 252, 254–257, 261, 302, 304, 311–313, 315, 316, 331, 370, 380, 441, 443, 444, 453, 454, 470, 471
Label, 27, 28, 34, 35, 38, 43, 50, 53, 75, 104, 136, 142–145, 147, 149, 150, 164, 167, 229, 231, 249, 252–256, 260, 277, 279, 282, 291, 298–300, 302, 310, 311, 358, 370, 378, 382, 419, 429, 435–437, 442
LALR, 75, 93, 107, 110, 112, 113, 127, 128, 145, 470, 471
LALR(1), 75, 93, 107, 110, 112, 113, 127, 145, 470, 471
Language
-Atoms, 50, 75
-Description, 113
-Idioms, 377
-Iff, 419
-Library, 182, 183, 221
-Macros, 374
-Node, 432
-Recognition, 128
-Theory, 409
Languages, 1–6, 15, 19, 22, 26, 46, 47, 50–53, 72, 73, 75, 76, 91, 92, 102, 112, 113, 128, 134, 140–142, 166–168, 171–175, 182, 186, 187, 190–192, 198, 202, 203, 206, 207, 209, 215, 221, 228, 229, 233, 241, 244, 248, 266, 279, 282, 283, 289, 297, 329, 332, 333, 335–337, 339, 341, 347, 396, 400, 406, 408, 410–413, 415, 417, 418, 422–424, 427, 428, 431, 434, 449
Languages
Assembly-, 434
Contextfree-, 412
Egular-, 412, 417, 418, 423
Formal-, 2, 392, 393, 395, 397, 399, 401, 403, 405, 407–409, 411–413, 415, 417, 419, 421, 423, 425, 427, 429, 431, 433
Free-, 412, 424, 429
Regular-, 412, 417, 418, 423
Structured-, 209
Large, 4, 9, 10, 19, 21, 23, 90, 104, 115, 116, 118, 133, 188,

202–204, 206, 211, 216, 220, 224, 260, 266, 270, 289, 291, 297, 327, 331, 333, 338, 347, 354, 355, 359, 397, 399, 418, 440, 451, 458, 459
Layout, 102, 187, 191, 224, 242, 269, 278–280, 286, 288, 451, 455, 456, 478
Left, 16, 29, 36, 37, 44, 58, 77–79, 85, 87, 88, 90–92, 95, 117, 119, 120, 123, 129, 146, 157–159, 164, 167, 168, 170, 172, 179, 189, 206–208, 210, 225, 231–234, 242, 249, 279, 300, 306, 341, 351, 355, 356, 362, 395, 403, 405, 408, 425, 426, 460, 462–464, 474
Left
-Bit, 29, 356
-Canonical, 77
-Child, 168
-Correct, 306
-Derivation, 77–79, 87, 426, 464
-Derivations, 426
-Derived, 85
-Implicit, 403
-Loop, 300
-Order, 189, 206, 225
-Parentheses, 405
-Produces, 77
-Recursion, 79, 119, 460, 462, 463
-Subscript, 280
Leftmost
-Bit, 29, 356
-Canonical, 77
-Derivation, 77–79, 87, 426, 464
-Derivations, 426
-Derived, 85
-Subscript, 280
Legacy, 188, 189, 191, 224, 347
Lemma
Pumping-, 72, 423, 429, 430
Level, 1, 3–5, 10, 19, 20, 23, 24, 26, 31, 51, 81, 92, 134, 135, 139, 141, 175, 179, 185, 199, 205, 214, 221, 227, 228, 241, 244, 260, 262, 266, 268, 270, 275, 282, 289, 291, 310, 314, 333, 335, 338, 346, 347, 412, 424, 447
Level
Chomsky-, 73
Levels, 19, 81, 175, 179, 289, 335, 424
Lex, 21, 30, 46, 50, 62–69, 71–73, 134, 137, 151, 167, 364, 424
Lexer, 66
Lexical, 11, 13, 14, 26, 30, 31, 47, 48, 50–52, 62, 63, 68, 70, 72–75, 90, 114, 115, 120, 124, 125, 154, 176, 197–199, 210, 285, 309, 337, 406, 412, 437, 440, 473
Lexical
-Analysis, 11, 13, 51
-Analyzer, 13, 25, 50, 51, 53, 55, 57, 59, 61–63, 65, 67, 69, 71, 73, 155, 267
-Atoms, 384
-Concerning, 25
-Related, 73
-Tokens, 13
-Values, 155
Code-, 62
Required-, 384
Scanner-, 267
Libraries, 4, 183, 220–222
Library, 8–10, 59, 66, 182–185, 202–204, 206, 216, 220–223, 225, 268, 270, 286, 315, 334, 349, 350, 360, 364, 375, 385, 386, 434, 448, 449, 451, 452, 455
Line
-Bounded, 415
-Control, 447
-Intermediate, 51
Messages-, 447
Tree-, 51
Linear
-Bounded, 415
Linkage, 193, 208, 220, 286
Linked, 22, 58, 63, 69, 159, 179, 185, 203, 204, 206, 213–215, 218, 220, 248, 266, 286, 306, 307, 312, 314, 315, 349, 374, 376, 398, 399, 437, 439, 440, 448–451
Linker, 8, 19, 20, 187, 291, 350, 357, 358, 361, 391, 448, 453, 455
Linker
-Algorithms, 453
-Combines, 187
-Connects, 309
-Fix, 448
-Loader, 10, 187
-Pass, 453, 454
-Printf, 452
Executable-, 449
Gcc-, 452
Linkers, 10, 448, 449, 451, 453, 455, 457, 459
Linking, 8–10, 19, 20, 73, 187, 220, 221, 268, 375, 384, 448–453, 455
Links, 25, 58, 126, 192, 197–199, 201, 211, 224, 228, 265, 315
Linux , 7, 8, 67, 69, 114, 204, 206, 216, 218–220, 225, 243, 264, 268, 287, 334, 335, 341, 353, 374, 382, 390, 424, 445, 455
Literal, 11, 27, 48, 51–53, 58, 64, 67, 117, 152
Live
-Analysis, 315, 326, 327
-Variable, 326
Iterative-, 326, 327
Liveness, 315
LL, 4, 5, 8–10, 22–25, 33, 46, 59, 60, 66, 75, 78, 79, 81–89, 92, 93, 100, 104, 106–113, 128, 135–137, 145, 147, 214, 218, 231, 232, 244–248, 258–260, 266, 302, 305, 342, 346, 363, 365, 372, 413, 434, 447, 460–463, 468, 470, 471
LL(0), 93, 113
LL(1), 75, 79, 81–87, 89, 93, 113, 135, 136, 460–463
LL(k), 78, 79, 93, 113, 135, 145
Load, 3, 8, 18, 25, 30, 42–44, 46, 91, 166, 173, 175, 185–188, 220–222, 243, 244, 264, 268,

270, 271, 276, 277, 281, 290, 291, 294, 296, 306, 315, 322, 333, 334, 346, 354, 360, 382, 386, 389, 436, 437, 442, 444, 448–451, 453, 455–459
Load
-Accumulator, 442
-Address, 450
-Format, 455
-Operations, 10, 185
-Value, 30
Absolute-, 3, 456
Extensible-, 455
Linker-, 10, 187
Linking-, 451, 455
Loader-, 3
Python-, 9
Redundant-, 320
Unnecessary-, 44
Loader, 3, 10, 25, 42, 43, 185–187, 202, 437, 448, 449, 451–453, 455–459
Loaders, 10, 448, 449, 451, 453, 455, 457, 459
Loading, 3, 166, 173, 175, 186, 188, 221, 268, 291, 296, 306, 322, 333, 354, 436, 449, 456
Local, 24, 80, 105, 134, 141, 144, 145, 174, 189–195, 198, 199, 201, 202, 205, 207, 226, 241, 243, 264, 280, 281, 283, 285, 286, 288, 314, 320, 325, 326, 378, 379, 381, 382, 442, 447, 449, 450, 453, 478
Logic
-Connectives, 394
-Instructions, 356
-Programming, 332
-Result, 368
-View, 451
Formal-, 392
Remarks-, 356
Logical, 5, 11, 21, 47, 103, 143, 159, 165, 187, 191, 225, 234, 242, 253, 298, 329, 356, 368, 394, 396, 399
Logical
-Connectives, 394
-Result, 368
-View, 451
Remarks-, 356
Lookahead, 57, 103, 106
LOOKBACK, 111, 112
Lookup, 33, 56, 59, 176–178, 180, 212, 214, 247, 362, 365, 366
Loop, 9, 19, 24, 27, 28, 31, 34, 38, 41, 44, 79, 134, 167, 172, 180, 184, 209, 224, 234, 236, 255, 256, 263, 281, 282, 287, 289, 298, 300, 312, 314, 321, 322, 324, 325, 329–331, 345, 347, 348, 370, 376, 379, 380, 401, 423, 440
Loop
-Loops, 379
-Unwinding, 324
Finished-, 379
OVER-, 346
Loops
OVER-, 346
Louden, 181, 225
LR, 75, 89, 92–96, 100, 102–108, 110, 112, 113, 118, 119, 127, 128, 136, 138, 145, 464–468, 470, 471
LR(0), 75, 93–96, 100, 102–107, 110, 112, 113, 118, 119, 136, 464–467, 470
LR(1), 75, 93, 102–104, 106–108, 110, 112, 113, 127, 136, 145, 464, 467, 468, 470, 471
LR(k), 89, 93, 103, 110, 113

M

Machine, 1–5, 9–11, 13, 15, 19–22, 24–26, 29, 30, 37, 38, 41–44, 46, 51, 63, 100, 104, 110, 112, 113, 115, 116, 129, 134, 136, 140, 183, 188, 192, 218, 227–229, 235, 238, 241, 242, 244, 246–249, 253, 260, 261, 265–274, 276, 278, 282, 285, 287, 289–291, 300, 302, 304, 308–311, 315, 320, 325, 333, 334, 338, 339, 341, 346, 347, 351, 355, 379, 412, 415, 418–421, 423, 427, 428, 430, 434–436, 447, 448, 450, 464–467, 470, 474
abstract, 38, 241, 248
Machines, 1, 2, 5, 134, 183, 188, 192, 227, 228, 235, 244, 246–248, 260, 272, 276, 287, 300, 341, 347, 421, 427, 434
Macro, 3, 4, 7, 11, 24, 66, 124, 217, 261, 270, 287, 308, 325, 350, 374, 384, 391, 434–437, 439, 441–447
Macros, 66, 217, 261, 270, 287, 308, 374, 384, 391, 434, 436, 441, 443–445
Major
-Operations, 159
Managed, 42, 182
Management, 146, 183, 189, 202, 203, 214, 218, 235, 240, 287, 309, 334, 459
Mapper, 30
Mapper
-Code, 13
-CodeGen, 13
-Front, 11, 228
Analysis-, 13, 51
Parser-, 11, 13, 228
Massalin, 327, 328, 331
Match, 42, 60, 62–67, 69, 81, 82, 85, 89, 123, 177, 178, 214, 300, 302, 338, 340, 342, 364, 392, 408, 431, 444, 475
Meaning, 5, 7–9, 11, 20, 21, 24, 25, 37, 38, 48, 50, 64, 73, 75, 139, 140, 142, 153, 154, 166, 175, 182, 207, 226, 239, 266, 267, 300, 358, 410, 411, 417, 431
Meanings, 24, 266, 410, 431
Member, 3, 42, 78, 90, 124, 142, 145, 152, 167, 180, 190, 193, 221, 234, 239, 242, 248, 254, 272, 274, 288, 300, 310, 325, 330, 360, 369, 371, 381, 382, 397, 399, 410, 418, 422, 423, 432, 434, 474

Memory
-Access, 346
-Allocation, 160, 188, 346
-Allocator, 225
-Based, 451
-Basic, 184
-Cycle, 309
-Dirty, 459
-Environment, 182
-Executable, 191, 451
-Hierarchy, 290
-Layout, 455
-Locations, 385
-Management, 287
-Organization, 456
-Space, 289
-Stack, 455
-Swap, 290
-Virtual, 184
CREATE-, 346
Dynamic-, 188
Free-, 187
Main-, 290, 459
Physical-, 458
Port-, 184
Process-, 191, 451
Shared-, 184
Virtual-, 184, 290, 451, 455, 456, 458
Meta, 27, 35, 48, 80, 138, 267, 412, 413
Meta
Primitives-, 64
Methods, 48, 78, 79, 89, 91, 102, 103, 142, 161, 168, 182, 185, 186, 188, 191, 204, 205, 208, 210, 221, 243, 268, 280, 282, 295, 310, 315, 320, 323, 325, 326, 331, 399, 412, 440
Methods
-Area, 243
-Vacuous, 404
Collector-, 205
Proof-, 404
Minic, 128, 133, 139, 160, 161, 249, 250, 255, 257, 258, 286–288, 303, 304, 307, 314, 348–351, 361, 362, 364, 365, 370, 373, 375–377, 384–387, 391
Mode
-Analysis, 393
Mathematics-, 393
Modelling-, 393
Modifications, 25, 47, 208, 232, 349, 475
Module, 4, 5, 10, 22, 25, 128, 137, 187, 220, 229, 270, 332, 335, 448–451, 453, 454
Module
-Definition, 453
Linked-, 450
Object-, 450, 451
Modules, 4, 5, 10, 22, 25, 128, 137, 187, 229, 270, 332, 335, 448–451, 453
Muchnick, 137
Multiple, 48, 49, 65, 128, 141, 153, 200, 221, 222, 225, 279, 333, 420, 421, 440, 442
Multiple
-Jobs, 186
-Next, 420
-Scanners, 52
-Ways, 335
Competence-, 335

N

NDFSM, 420–422, 430
Nested, 64, 192, 196, 199, 309
Noam Chomsky, 2, 412, 415
Normal, 247, 355, 412, 414, 426, 451
Notation, 47, 85, 142, 154, 161, 164, 172, 173, 227, 232, 233, 235, 238, 244, 250, 263, 267, 273, 331, 345, 367, 397, 473
Notation
Equivalence-, 395
Implication-, 395
Operation-, 394
Polish-, 227, 232, 270
Quadruple-, 238
Tuple-, 237
Notations
Equivalence-, 395
Implication-, 395
Operation-, 394
Notes, 38, 72, 83, 93, 97, 103, 137, 142, 151, 154, 155, 157, 217, 218, 238, 248, 394, 395, 397, 398, 400, 406, 410, 413, 415, 417, 422, 425, 450
NT, 35, 77, 78, 81, 85–88, 94, 95, 110, 116, 138, 239, 413, 415, 422, 425, 426, 429, 460, 463
Nullable, 86, 87, 138

O

Object, 1, 5, 7, 9, 19, 22, 48, 63, 73, 128, 134, 141, 186, 187, 190, 205, 206, 208, 220–222, 228, 229, 260, 266, 267, 270, 286, 329, 332, 334, 335, 339, 349, 358, 375, 386, 403, 409, 432, 433, 436, 437, 440, 448–451, 455
Objectives, 349
Operand, 3, 38, 46, 73, 131, 143, 158, 160, 166, 217, 233–235, 237, 239–241, 243, 244, 249, 267, 273–276, 306, 308, 309, 317, 328, 329, 354, 355, 360, 369, 390, 435, 437, 456, 457, 474
Operand
-Pointer, 388
-Quoted, 358
Equ-, 358
Operating
-Environment, 287
-System, 48, 182, 183, 185, 186, 216, 225, 287, 309, 339–341
-Systems, 225, 339–341
Disk-, 186
Linux-, 287
Synchronized-, 165
Target-, 287

Operation
-Actions, 267
-Insert, 209
-Let, 398
-Notations, 394
-Semantics, 142
-Stack, 233
Binary-, 409, 411
FPU-, 384
Point-, 308
Set-, 398
Transformations-, 321
Operator
-Grammar, 90
-Precedence, 75, 90, 92
-Symbols, 267
Table-, 52
Operators, 27, 47, 48, 50, 51, 53, 56, 64, 75, 91, 92, 121, 166, 175, 233, 277, 337, 394
Optimization, 8–12, 19–23, 26, 37, 44, 48, 113, 114, 134, 135, 141, 161, 182, 200, 227, 228, 235, 238, 239, 242, 244, 253, 262, 264, 267–270, 273, 274, 287, 289–291, 295, 300, 302, 308, 310–312, 314–316, 320, 323, 325, 327, 329–331, 346, 347, 349, 350, 382, 384, 391
Optimization
About-, 310
Architectures-, 391
Assembler-, 13
Code-, 18, 289, 310, 311, 313, 315, 317, 319, 321, 323, 325, 327, 329, 331
Dependent-, 18, 268, 269, 271, 273, 275, 277, 279, 281, 283, 285, 287, 289, 291, 293, 295, 297, 299, 301, 303, 305, 307, 309
Hole-, 320
Structural-, 310, 323
Optimize, 5, 18, 19, 22–24, 26, 44, 58, 146, 231, 239, 244, 261, 265, 269, 273, 289, 310, 312, 323, 327–331, 341, 350, 351
Optimized, 5, 18, 19, 23, 24, 26, 58, 146, 231, 265, 273, 289, 312, 323, 329, 341, 350, 351
Optimizer, 261, 269, 289, 310, 323, 327, 328, 330, 331
Optimizers, 310, 327, 331
Order
-Relations, 402
Ordering, 268, 275, 302
ORG, 435, 438
Organization, 113, 127, 165, 182, 189, 308, 456
Outputs, 9, 10, 13, 42, 72, 114, 119, 125, 127, 227, 229, 240, 249, 261, 265, 350, 354, 367, 368, 370, 384, 387, 432, 445
Overloading, 166, 173, 175

P

Page, 19, 114, 128, 223, 225, 290, 330, 333, 349, 424, 447, 455, 458, 459
Pages, 114, 128, 223, 447, 455, 458, 459
Parameter, 22, 142, 145, 177, 178, 182–184, 187, 189–191, 196, 197, 199, 206, 207, 222, 223, 225, 241, 242, 247, 248, 249, 276, 280, 283–286, 288, 294, 300, 345, 383
Parameters, 142, 178, 182–184, 187, 190, 191, 196, 197, 199, 206, 207, 222, 223, 225, 241, 242, 249, 276, 280, 283–286, 288, 300, 345, 383
Parse, 11, 13, 16, 18, 25, 26, 28, 30, 31, 34, 36, 37, 39, 41, 46–48, 50–53, 61–63, 68–71, 73, 75, 76, 78–84, 88–96, 100, 102–104, 106, 108, 110, 112, 114–121, 123–125, 127, 128, 131–143, 145, 150–154, 157–159, 164, 165, 178, 186, 223, 227, 229, 230, 232, 233, 235, 239, 240, 252, 260, 262, 264, 265, 267, 280, 321, 329, 335, 351, 367, 370, 384, 426, 465, 467–469, 471–474
Parser
LL(1), 81, 83, 84, 89
LL(k), 93
LR, 75, 100, 103, 127, 128, 138
LR(0), 94–96, 100, 103
LR(1), 75, 103, 104, 106, 110, 127, 145
LR(k), 89, 93
Parser
Analysis-, 13, 51
Automatic-, 112
Based-, 152, 252
Basic-, 138
Bison-, 68
Buggy-, 80
Cloned-, 128
Code-, 269
Complete-, 36
Constructing-, 426
Descent-, 16, 26, 34, 75, 79, 80, 229, 252, 267
Down-, 267
Driven-, 106, 138
Example-, 16
FSM-, 50, 75
Full-, 62
Generate-, 128
Generated-, 115, 127
GLR-, 127
Known-, 150
LALR-, 128
Multiple-, 128
Oriented-, 128
Perl-, 128
Precedence-, 75, 89
Prefix-, 140
Program-, 13, 230
Project-, 73
Real-, 153
Reduce-, 89, 91, 102
Scanner-, 11, 52, 228
Sensitive-, 141
Simplified-, 18
SLR-, 100

Symbols-, 89
Tree-, 230
Type-, 82, 110
Parsers
Cloned-, 128
Constructing-, 426
Generate-, 128
GLR-, 127
LALR-, 128
Multiple-, 128
Precedence-, 89
Real-, 153
Reduce-, 89
Parsing
Actual-, 110
Basic-, 75
Bison-, 75
Control-, 90
Current-, 150
Discussing-, 425
Down-, 75–79, 89, 460
Expression-, 142
General-, 127
Incomplete-, 89
Known-, 50, 75
Make-, 92
Minimal-, 164
Modified-, 85
Precedence-, 89–92
Reduce-, 89
Right-, 474
PASCAL, 6, 186, 190, 209, 227, 241, 282
Pass
Goto-, 438, 443, 454
Linker-, 453, 454
Passing-, 206
Processing-, 443, 444
Passed, 4, 35, 70, 71, 120, 141, 154, 182, 203, 207, 208, 218, 223, 224, 257, 280, 283, 285, 300, 346, 447
Passing, 5, 145, 154, 182, 189, 206, 208, 347, 385, 474
Pattern, 62, 64, 73, 298
Patterns, 62, 73, 298
PDA, 50, 75, 79–82, 89, 92–95, 115, 118, 138, 183, 415, 424, 427–429, 431
Pedagogic, 26, 269
Performed, 2, 43, 73, 74, 94, 119, 128, 227, 240, 254, 265, 338, 474
Perl
-Documentation, 336
-Especially, 336
-Execute, 336
-Extension, 128
-Inserts, 336
-Internals, 338
-Language, 62, 128, 180, 307, 335
-Looks, 336
-Object, 128
-Parser, 128
-Program, 223, 264, 336, 338
-Programs, 223
-Provides, 266
-References, 338
-Scalars, 338
-Script, 302, 303, 307, 336
-Source, 338
-Uses, 338
-Values, 338
Compiled-, 338
Complete-, 307
Learned-, 336
Phases, 1, 10–13, 20, 21, 24, 30, 33, 46, 47, 50, 51, 114, 140, 142, 153, 209, 211, 212, 229, 239, 269, 412, 434, 447, 453
Physical, 24, 159, 204, 309, 379, 455, 456, 458, 459
Pieces, 429, 434
Pigeonhole
-Principle, 409
Generalized-, 409
Pike, 127, 137, 176, 348, 349
Pipelining
Instruction-, 300
Plet, 37, 39
Pointers, 164, 171, 172, 197, 202, 205, 206, 213, 215, 226, 238, 264, 266, 279, 280, 338, 381, 439, 440, 456
Polish
-Logician, 232
-Notation, 227, 232, 233, 235, 244, 263, 267, 270, 367
-Reverse, 227
Reverse-, 227, 233, 235, 244, 267, 270, 367
Popular, 22, 175, 216, 238, 270, 332, 347, 350, 440
Populated, 338
POSIX, 44, 48, 216, 445
Powerful, 62, 79, 83, 128, 141, 332, 335, 348, 404, 406, 427, 441
Pplus, 16, 17, 36, 39, 80, 148, 150, 230, 232, 235, 240, 464
Precedence, 63, 75, 89–92, 103, 121, 123, 127, 128, 136, 173, 179, 417, 425
Precedence
-Parsing, 90
-Relations, 91
Operator-, 90
Precision, 121, 138, 357, 360, 368, 388
Prefix
-Associated, 94
-Ending, 94
-Notation, 232
-Parser, 140
-Property, 92
-Rev, 424
-Set, 94
-Viable, 93
Name-, 336, 337
Proper-, 410
Specified-, 180
Valid-, 136
Viable-, 93–97, 104, 136, 465

Preprocessor, 19
Principle
Drawer-, 409
First-, 405
Pigeonhole-, 409
Second-, 405
Problem
-Consider, 102
-Modelling, 393
Real-, 393
Procedure, 52, 133, 141, 186, 187, 191, 192, 199, 205–207, 242, 249, 265, 267, 268, 283–285, 421, 475, 478
Procedures, 268, 283
Processing
Art-, 226
End-, 350
Exception-, 309
Further-, 31, 229, 270
Later-, 131
Macro-, 442–444
Pre-, 19, 20, 443
Statement-, 8
String-, 308
Text-, 332, 335
Processor
Bit-, 390
CISC-, 188
Commercial-, 248
Full-, 290
Hardware-, 248
Initial-, 327
Language-, 392
Macro-, 7, 11, 441, 444, 445
Main-, 308
Modern-, 249, 284, 300
Most-, 322
Motorola-, 328
Normal-, 320
Particular-, 253, 265, 327, 330
Pipeline-, 301
Pipelined-, 328
Point-, 183, 189
Pre-, 19
RISC-, 188, 282
Script-, 8
Separate-, 301
Series-, 218
Set-, 113
Single-, 300, 309
Target-, 271, 272, 282, 289
Typical-, 283
Processors, 7, 8, 134, 188, 218, 238, 249, 268, 274, 279, 282, 284, 296, 300, 301, 320, 322, 324, 328, 390, 392, 441, 444
Processors
Macro-, 7, 434, 435, 437, 439, 441, 443–445, 447
Production, 27, 35, 36, 48, 76–79, 81, 83, 85–87, 89, 90, 93–96, 100, 103, 104, 109, 110, 112, 116, 117, 138, 151, 153–155, 159, 228, 267, 413, 415, 422, 424, 426, 460, 464, 473, 474
Products, 138, 406
Programmer, 3, 5, 21, 24, 29, 44, 46, 70, 102, 166, 167, 172, 175, 185, 186, 202–204, 208, 209, 216, 221, 267, 269, 289, 322, 329, 334, 336, 339, 354, 412, 414, 434, 436, 441, 444, 453, 456, 474
Programming, 2–6, 8, 15, 22, 24, 26, 46–48, 50, 52, 62, 67, 70–73, 75, 112, 113, 140–142, 164, 172, 173, 175, 185, 186, 206, 208, 209, 233, 234, 238, 266–268, 283, 289, 325, 332–335, 337, 339, 341, 347–349, 390, 392, 396, 400, 410–415, 418, 424, 431, 434, 449
Programming
-Environment, 127, 176, 348, 349
-Guide, 391
-Language, 225, 331, 332
Logic-, 332
Structured-, 22, 310
System-, 225, 391
UNIX-, 127, 176
Unix-, 348, 349
PROLOG, 4, 248, 249, 332, 339–341
Proof
-Methods, 404
-Technique, 403, 405, 406
Direct-, 404
Trivial-, 404
Pruning, 327, 331
Pumping
-Lemma, 72, 423, 429, 430
Purposes, 170, 224
Python, 5, 8, 9, 11, 22, 26, 72, 73, 128, 207, 283, 347, 399

Q

Quick, 8, 218, 228, 290, 327, 328, 331–333

R

RDP, 26, 34, 35, 37, 39, 75, 79–81, 90, 143–147, 230, 232, 235–237, 240, 252, 463, 464
RE, 13, 53, 55, 63, 64, 417–420, 422, 424, 426, 431, 463
READS, 112
Real
-Constants, 242
-Division, 242
Strings-, 242
Recognizing, 36, 423, 429, 431
Record
Activation-, 182, 196, 287, 476, 478
Records-, 453
RLD-, 453
Records
Activation-, 182, 196, 476
RLD-, 453
Recovery, 71, 74, 81, 116, 123, 202
Recursion
Allow-, 283
Allows-, 208
Left-, 79, 119, 460, 462, 463

Performing-, 445
Right-, 79, 119
Recursive, 16, 34, 36, 39, 48, 79, 80, 87, 90, 144, 145, 167, 171, 172, 185, 186, 188, 192, 194, 229, 248, 252, 267, 283, 406, 407, 424, 426, 431, 475, 477, 478
Recursive
-Algorithms, 406
-Arithmetic, 90
-Call, 39, 283, 407
-Calls, 283
-Check, 171
-Definition, 85, 87, 403, 417, 431
-Descent, 39, 407
-Enumerable, 415
-Form, 36, 48
-Function, 39, 145, 185, 188, 194, 283, 406
-Functions, 39, 185, 188, 406
-Grammar, 79
-Procedures, 186, 267, 475
-Production, 424
-Rule, 79, 426
-Rules, 426
-Subroutines, 186
-Type, 171, 172
-Types, 171
-Way, 413
Reduce, 46, 62, 77, 78, 88–91, 94, 95, 97, 100–110, 112, 116, 118, 119, 121–124, 127, 128, 134, 136, 138, 151, 153, 158, 199, 204, 206, 209, 212, 217, 274, 276, 287, 289, 316, 321, 323, 468
Reduced
-Considerably, 212
-Grammar, 78
-Grammars, 78
-Instruction, 270
Reduction, 19, 23, 76–78, 89, 102, 104, 120, 121, 124, 127, 128, 138, 151, 173, 229, 289, 291, 320, 321, 323, 471, 475
Reduction
Correct-, 104
First-, 321
Legal-, 121
Possible-, 127
Right-, 78
Rightmost-, 77, 78
Small-, 475
Space-, 289
Strength-, 320, 321
Time-, 289
Redundant
-Computations, 141
-Expression, 316
-GoTo, 320
-Instruction, 44, 291, 295
-Instructions, 44, 291
-Load, 320
-Parentheses, 164
-Rules, 78
-Sub, 323
Reference
-Arguments, 207
-Count, 204–206
-Counter, 204
-Counting, 205, 206
-Counts, 205, 206
-Data, 316
-Denoting, 267
-Evaluation, 207
-Generated, 253
-Implicit, 209
-Information, 312, 313
-Item, 153
-Later, 312
-Manual, 391
-Method, 207
-Numbers, 238
-Point, 193
-Semantics, 208
-Simultaneously, 359
-Text, 137
-Therefore, 207
-Value, 207
Address-, 197
Array-, 280, 320
Descriptor-, 383
Existing-, 205
EXPR-, 261
External-, 453
Forward-, 205, 436, 437
Function-, 208
GCC-, 260
Hold-, 338
Local-, 449, 450
Memory-, 49, 390
Next-, 312–316
Optimization-, 391
Perl-, 338
Ready-, 167
Sequence-, 238
Set-, 359, 391
Standard-, 137
Structure-, 448
Symbolic-, 455
Tuple-, 315
Updating-, 206
Variable-, 200, 277
References, 49, 62, 159, 200, 205, 209, 253, 277, 280, 312, 319, 320, 338, 382, 383, 399, 436, 437, 448–450, 455
References
-Generated, 253
Array-, 280, 320
Descriptor-, 383
Forward-, 436
Hold-, 338
Local-, 450
Perl-, 338
Symbolic-, 455
Variable-, 200, 277
Regarding, 1, 103, 141, 339
Registers, 18, 29, 30, 42, 44, 46, 113, 141, 187, 188, 192, 205, 216, 218, 238, 241, 247, 248, 267–272, 274, 276–278, 283, 285, 286, 289, 291, 292, 294–298, 300, 304, 306, 316, 325, 327, 328, 355, 359, 360, 378, 388, 390

Registers
Assumed-, 294
Basic-, 248
Control-, 192
Coupled-, 274
CPU-, 18, 113, 187, 188, 216, 218, 238, 269–271, 274, 291
Display-, 46
Eax-, 355
Extended-, 192
Fast-, 274
Fewest-, 297
Five-, 241
FPU-, 360
General-, 272
Integer-, 360
Length-, 188
Machine-, 44, 46, 247, 278, 304
Numbered-, 30
Point-, 360, 388
Processor-, 267, 289, 316, 325, 327, 328
Purpose-, 188, 271, 272, 274, 296
Saving-, 285
Sized-, 390
Use-, 276, 292, 296, 298
Regular
-Expression, 13, 50, 51, 53, 63, 69, 72, 73, 79, 128, 135, 266, 337, 338, 364, 392, 407, 412, 414, 417, 424, 430, 431
-Expressions, 13, 50, 51, 53, 63, 69, 72, 79, 266, 337, 392, 407, 412, 414, 417, 424, 431
-FSM, 50, 75
-Grammar, 61, 63, 73, 417, 422
-Iff, 422
-Language, 51, 53, 73, 93, 412, 414, 417–419, 422, 423
-Languages, 51, 53, 412, 417, 418, 423
-Operations, 418
-Patterns, 62
-Shift, 356
-Table, 75
-Way, 344
Relation
Ary-, 401, 402, 431, 432
Binary-, 401, 402, 431, 432
Callee-, 476, 477
Consider-, 140
Equivalence-, 402, 432
Further-, 339
Identity-, 402
INCLUDES-, 112
New-, 111
Numbers-, 397
Order-, 402
Ownership-, 339
Precedence-, 90, 91
Range-, 433
Relationships-, 4, 48, 140, 419
Specify-, 339
Syntactical-, 48
Useful-, 111
Way-, 412
Relations
Binary-, 401
Callee-, 476, 477
Consider-, 140
Equivalence-, 402
Further-, 339
Numbers-, 397
Order-, 402
Ownership-, 339
Precedence-, 90, 91
Relationships-, 4, 48, 140, 419
Specify-, 339
Syntactical-, 48
Way-, 412
Relationship, 4, 5, 48, 50, 62, 140, 145, 339, 345, 412, 419, 431, 476, 477
Relative
-Address, 160, 186, 222, 309, 457
-Addresses, 160, 186, 222, 457
-Easy, 241
-Index, 237
-Jump, 30
-Positioning, 383
-Small, 9, 320
-Strict, 334
Reloading, 291
Relocating
-Loader, 457
Relocation, 448–450, 453, 456
Representation, 3, 4, 25, 48, 51, 53, 73, 118, 138, 139, 143, 145, 149, 150, 160, 161, 227, 229–233, 235, 237, 249, 260–262, 265, 268, 269, 272, 295, 300, 331, 338, 348, 349, 367, 370, 399, 408, 431, 473
Representation
-Output, 161
-Reflexive, 401
Digraph-, 401
Graphical-, 401
Intermediate-, 139, 161, 269, 270, 348
Syntax-, 267
Rescheduling, 300
Reserved, 3, 47, 48, 73, 202, 205, 216, 286, 287, 289
Resolution, 121, 138
Resolved, 75, 103, 105, 108, 112, 121, 127, 158, 176, 201, 272, 425, 436, 457
Restore, 189, 190, 193, 208, 285, 325
Results, 31, 37, 73, 76, 109, 121, 134, 152, 175, 186, 189, 204, 220, 235, 265, 275, 277, 283, 288, 289, 292, 300, 307, 315, 317, 323, 329, 338, 355, 356, 360, 408, 440, 451
Returns, 39, 40, 52, 66, 115, 116, 119, 151, 172, 176, 177, 192, 193, 203, 204, 206, 207, 217, 218, 236, 237, 240, 283, 285, 328, 342, 343, 371, 383, 476
Reverse
-Polish, 227, 233, 235, 244, 267, 270
RG, 63, 417
Richard, 127, 137, 225, 265, 331

Rightmost
-Bit, 356
-Derivation, 78, 89, 92, 93, 104, 426, 464
-Derivations, 92, 93, 464
-Reduction, 77, 78
-Subscript, 279
-Symbol, 95
RISC, 22, 184, 188, 270, 282
RL, 417, 423
RLD, 453, 454
Routine
Service-, 323
RPN, 146, 149, 150, 152, 161–163, 227, 233–237, 240, 249–259, 264, 267, 270, 271, 302, 303, 306–308, 315, 349–351, 367, 368, 370, 384
Run
-Program, 216
Static-, 188
Running, 23, 45, 69, 141, 166, 182, 183, 186, 187, 199, 205, 206, 212, 216, 220, 285, 333, 342, 435, 445, 448, 451, 458, 475

S

Sample, 30, 158, 223, 230, 231, 240, 248, 261, 265, 359, 368
Scanner, 11, 13–16, 24–26, 30, 31, 33–37, 41, 46–48, 50–56, 58–63, 67–73, 75, 92, 114, 120, 131, 133, 134, 137, 139, 141, 142, 151, 159, 170, 176, 186, 209, 213, 215, 228, 229, 239, 259, 267, 269, 329, 361, 364, 371, 384
Scanners, 50, 52, 61, 62, 72, 137
Scheme, 1, 6, 8, 25, 165, 185, 186, 199, 203, 228, 237–239, 270, 277, 278, 282, 283, 291, 292, 294, 295, 297–300, 316, 317, 456, 473, 475
Scheme
Numbering-, 310, 316
Schemes, 1, 6, 185, 228, 238, 278, 283, 297, 299, 300
Scope, 22, 141, 159, 160, 163, 182, 207–209, 224, 265, 309, 337
Scripting, 8, 332
SDD, 157, 471, 472
SDT, 165, 473
Search
-Information, 226
-Tree, 211, 349, 362
Binary-, 211, 349, 362
SECD, 227, 248
Selected, 26, 61, 84, 86, 228, 241, 268, 273, 332, 348, 349, 398, 423, 439
Semantic
-Analysis, 11, 13, 51, 160, 213, 329, 384
-Analyzer, 18, 62
-Concerning, 25
-Phase, 37, 269
-Prod, 145
-Related, 73, 267
-Rules, 158
-Stack, 139, 143, 145, 150
-Stacks, 139, 143, 145
-Value, 140
Formal-, 173
Imperative-, 140
Stack-, 150
Synchronized-, 143, 145
Tree-, 62
Semantics, 42, 43, 140, 142, 145, 152–154, 166, 173, 180, 202, 208, 221, 250, 270, 271, 289, 300, 327, 331, 344, 345, 391
Semantics
-Value, 140
Formal-, 173
Imperative-, 140
Sentence, 21, 48, 50, 72, 73, 75–79, 83, 85, 87–89, 115, 116, 119, 125, 136, 139, 140, 154, 174, 239, 267, 393, 410, 412, 413, 418, 425, 426
Sequence, 2–4, 13, 18, 31, 37, 38, 44, 73, 76, 78, 85, 117, 140, 153, 164, 171, 188, 205, 227, 237, 238, 244, 248, 249, 260, 266, 267, 269, 272, 273, 278–280, 282, 287, 289, 291, 295, 308–310, 316, 327, 328, 331, 343, 346, 358, 378, 384, 395, 396, 406, 407, 409, 410, 419, 422–424, 433–435, 441
Sethi, 137, 165, 181, 225, 331
Sets
Arbitrary-, 397
Basic-, 402
Command-, 256
Disjoint-, 85
Finite-, 427
FOLLOW-, 108, 111, 112, 463, 468
Gcc-, 206, 225
Implementing-, 399
Infinite-, 404, 408
Instruction-, 47, 369
Item-, 100, 103–107, 110, 112
Live-, 326
Local-, 326
Mixed-, 399
Represent-, 398
Specify-, 397
Trampoline-, 202, 226
Sets
Descriptors-, 383
Infinite-, 407
Special-, 397
String-, 410
Shared
-Libraries, 222
-Library, 222
Sharing, 207, 208, 478
Shift
-Actions, 104
-Reduce, 469
Parsing-, 89
Rule-, 469
Shifter, 37, 308
Short, 8, 32, 64, 66, 167, 183, 185, 190, 208, 226, 236, 249, 274, 275, 277, 279, 287, 295, 325, 327–329, 362, 392, 401, 410, 436

Showing, 48, 81, 201, 406, 421, 427, 431, 453, 478
Siblings, 153
Signature, 173, 181
Simple
-Approach, 327
-Arrangement, 234
-Array, 439
-Assembly, 216
-Bottom, 90
-Computer, 456
-Control, 412
-Description, 346
-Embedded, 445
-English, 310
-Example, 44, 235, 243, 274, 445
-Examples, 235, 274
-Expedient, 127
-Expression, 239, 414
-FSM, 419
-Function, 180, 291
-Functional, 180
-Generative, 412
-Grammar, 116, 117, 125, 473
-Heuristics, 312
-High, 26
-Idea, 441
-Idiom, 273
-Interpreted, 348
-Interpretive, 175
-Jump, 301
-Language, 21, 26, 30, 42, 46, 47, 71, 72
-Look, 27, 106
-Looks, 27
-Macro, 441
-Meanings, 431
-Means, 105
-Memory, 186, 188
-Method, 42, 79, 143, 256
-Methods, 79
-Operation, 131
-Optimization, 316
-Parser, 34
-Path, 433
-Perl, 336
-Phrase, 51, 76, 77, 425
-Program, 31, 34, 38, 216, 264
-Representation, 399
-Signed, 338
-Situation, 186
-Solution, 102
-Statements, 234
-Subset, 275
-Syntax, 332, 345
-Target, 26, 185
-Test, 460
-Translator, 26, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49
-Type, 170, 171
-Types, 171
-Virtual, 346
-Way, 43
Simplified, 8, 9, 16, 18, 55, 58, 104, 128, 224, 268, 270, 348, 384, 426, 443
Simulator, 26, 44–46
Simulator
Machine-, 45
Single
-Abstract, 241
-Active, 477
-Argument, 178, 328
-Assignment, 260
-Bit, 204
-Byte, 247
-Component, 168
-Copy, 448
-Executable, 448
-Instruction, 320, 327
-Job, 186
-Load, 187
-Module, 448
-Null, 417
-Operand, 329
-Pass, 254, 436, 474
-Processor, 300, 309
-Program, 186, 283
-Quotes, 27
-Sequential, 301
-Source, 8, 12
-Stack, 188
-Step, 386, 387, 389
-Stepping, 386, 387, 389
-Sum, 151
-Symbol, 103
-Target, 12
-Tree, 76
-Value, 336
Singly, 83, 349, 399, 440
Situations, 8, 12, 22, 56, 75, 184, 290, 323, 330, 392, 394
Sized, 4, 22, 24, 139, 151, 154–159, 164, 203, 204, 224, 225, 257, 280, 390, 418, 451, 472–474
SLR, 75, 93, 100, 104, 106, 108, 110, 112, 113, 136, 464, 467, 468, 470, 471
Space
Save-, 322
Spaces, 9, 33, 49, 53, 176, 204, 343, 345
Specification, 2, 24, 26, 27, 30, 63, 72, 73, 81, 94, 121, 132, 136, 137, 139, 140, 142, 151, 173, 179, 228, 242, 267, 279, 339, 347, 351, 411, 418, 475
Specification
Grammar-, 119, 414
Input-, 117
Specifications, 63, 72, 73, 94, 139, 173, 242, 475
Stacking
Implicit-, 143
Stacks, 143, 145
Starting
-Program, 387
State
Compiling-, 293
PDA-, 428
REF-, 261
Statements-, 4
Tokens-, 50, 75
Statements
Access-, 384

Action-, 384
Assignment-, 473
Assignments-, 249
Check-, 174
Compiling-, 293
Consider-, 53
Control-, 27
Debugging-, 69, 124
Declaration-, 257
Define-, 115
ELSE-, 103
False-, 253
FORTRAN-, 52
Generation-, 144
Intervening-, 252
Language-, 18, 73, 441
Larger-, 297
Processor-, 55
PROG-, 34
Program-, 34
RTL-, 261
Second-, 9
Sequenced-, 260
Tokens-, 50
True-, 253
Static
-Allocation, 186
-Char, 367
-Defined, 186
-Int, 208, 243
-Libraries, 222
-Library, 221, 222, 452
-Link, 197–199, 201, 202, 211, 224, 226, 242, 451, 452
-Linking, 451, 452
-Links, 197–199, 201, 211, 224
-Manner, 191
-Memory, 187
-Nesting, 197, 309
-Run, 188
-Semantic, 141, 142, 166
-Semantics, 142, 166
-Single, 260
-Storage, 186, 475
-Struct, 132
-Typed, 172
-Variables, 378
-Void, 243, 264, 265, 334
-Whenever, 477
Steps
Algorithm-, 36
Basic-, 70
Compilation-, 131, 159
Construction-, 240
Detailed-, 110
Further-, 268
General-, 384
Generation-, 182
Insertion-, 39
Linker-, 291
Optimization-, 316, 350
Parsing-, 142
Particular-, 123
Processing-, 19
Synthesis-, 227
Writing-, 26
Storage
-Allocation, 185, 475
-Classes, 208
-Off, 290
Strength
-Reduction, 321
Strings
-Accepted, 424
-Beads, 244
-Boolean, 432
-Bxy, 407
-Consisting, 72
-Diagnostics, 447
-Differ, 282
-Except, 407
-Match, 64
-Over, 405, 409–411
-TERM, 223
-Vertex, 432
Argument-, 144
Character-, 147, 282
Five-, 430
Flow-, 244
Growing-, 186
Input-, 10
Length-, 86, 186, 475
Matches-, 62
New-, 423
Prompt-, 183, 223
Required-, 235
Text-, 4, 62
Thread-, 244
Typical-, 424
Unmatched-, 66
Structure
-Becomes, 206
-Below, 167
-Containing, 260, 265
-Generated, 262
-Implementation, 391
-Initiated, 361
-Language, 173, 182, 190, 196–199, 209, 210, 212, 213, 215, 377, 475
-Languages, 173, 182, 190, 198, 209, 215
-Level, 412
-Making, 73
-Members, 381
-Programming, 266
-Provides, 228
-Referenced, 448
-Representing, 265
-Themselves, 412
-Typically, 206
-Used, 134, 260, 264, 317, 439, 443
Access-, 381
Basic-, 224
Block-, 173, 209, 210
Complex-, 173
Control-, 412
Data-, 37, 48, 59, 73, 134, 151, 167, 179, 186, 206, 213, 216, 221, 224, 230, 260, 265–268, 278, 283, 286, 317, 320, 338, 361, 399, 437, 439, 443, 448

Structure (continued)
Declared-, 281
Discrete-, 331
Dynamic-, 406
Else-, 426
Execution-, 475
Flow-, 325
Grammatical-, 25, 139, 267
Hierarchical-, 63
Implementation-, 228
Lexical-, 406
Linked-, 206
List-, 211, 464
Nested-, 64, 210
Nesting-, 197
Node-, 58, 146, 170, 179, 264, 314, 315
Organizational-, 308
Overall-, 135, 323
Particular-, 239
Sentence-, 73
Stack-, 283
Syntactic-, 50, 75, 154
Tree-, 58, 168, 231, 261, 349
Tuple-, 314
Usually-, 279
Well-, 210
Structured
-Language, 173, 182, 190, 196–199, 209, 210, 212, 213, 215, 377, 475
-Languages, 173, 182, 190, 198, 209, 215
Block-, 173, 209, 210
Well-, 210
Structures
Data-, 187, 278, 339, 340
Subroutine
Code-, 243
External-, 449
Language-, 246
Recursive-, 186, 188
Special-, 188
Super
-Block, 219
-Employs, 226
-Instructions, 347
-Optimizer, 331
SVs, 338
Symbol
-ADDRS, 442
-Addresses, 434, 436
-Appearing, 155
-Array, 453
-Arrive, 440
-Class, 159
-Control, 145
-Data, 295, 440
-Debugging, 211
-Denotes, 413
-Denoting, 104, 267
-Depend, 152
-Derivable, 425
-Dictionary, 450, 453
-Expression, 475
-Follows, 86, 97
-Indicates, 174
-Information, 7
-Int, 154
-Label, 252, 253
-Labels, 253
-List, 32, 33
-Look, 106, 212
-Lookup, 212
-Name, 439, 440
-Names, 439
-Node, 32, 33, 213
-Nodes, 32, 33
-Num, 473
-Occurring, 267
-Parsers, 89
-Pattern, 441
-Positions, 430
-Precede, 159
-Procname, 257
-Put, 409
-Read, 147
-References, 455
-Representation, 265, 295
-Representations, 295
-Represents, 152
-Specify, 152
-Stands, 415
-Supplied, 115
-Tab, 48, 50, 52, 53, 58–62, 68, 71, 131, 139, 159, 160, 167, 170, 173, 176–178, 180, 210, 213, 215, 313, 315, 317, 348, 361, 362, 434, 436, 439, 450
-Table, 7, 32, 48, 50, 52, 53, 58–62, 68, 71, 129, 131, 134, 139, 142, 152, 158–161, 163, 167, 170, 173, 176–180, 186, 209–215, 224, 257, 270, 313, 315, 317, 319, 348, 349, 351, 361, 362, 364, 370, 384, 391, 434, 436, 437, 439, 440, 442, 450, 453, 475
-Tables, 58, 131, 159, 180, 440
-Token, 115
-Type, 159
-Ultimately, 85
-Use, 153, 267, 413
-Used, 267, 413
-Value, 158
-Values, 158
-Varlist, 159
Symbols
Action-, 143, 152
External-, 453
Operator-, 267
Synchronized
-Operating, 165
-Semantic, 143, 145
Syntax, 11, 13, 15, 25, 37, 51, 62, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 101, 103, 105, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153–155, 157, 159, 161, 163, 165, 227, 239, 240, 267, 270, 275, 390, 415, 471, 473
Syntax
-Analyses, 141
-Analysis, 15, 51, 74, 75, 140
-Analyzer, 47, 48, 63

-Analyzers, 47
-Construct, 52
-Definition, 157
-Devoid, 233
-Directed, 157
-Employed, 128
-Errors, 15
-Highlighter, 164
-Sub, 337
-Tree, 15, 18, 63, 73, 81, 158, 227, 233, 235, 239, 240, 260, 265, 297, 414, 415
Syntax
-Trees, 18, 233, 414
-Used, 275
Abstract-, 73, 227, 233, 239, 260, 265, 415
Additional-, 345
Aspects-, 152
Assembly-, 275
Constitute-, 155
Detect-, 15
Detected-, 52
Expression-, 128
First-, 140
Intel-, 275, 276, 353, 390
Java-, 334
Language-, 221
Maps-, 18
Official-, 353, 390
Operand-, 276
Possible-, 474
Program-, 235, 240
Words-, 75
Yacc-, 136
Synthesized
-Attribute, 151, 154
-Attributes, 139, 154, 155
System
-Call, 182, 216–220, 287, 382
-Calls, 182, 216–220, 287, 382
-Environment, 183, 222
-Environmental, 183, 222
-Interface, 48
-Programming, 225, 391
-Services, 183
-Storage, 290
Algebraic-, 410
CPU-, 183
File-, 290
Linux-, 216, 218–220, 287, 382
Operating-, 48, 182, 183, 185, 186, 216, 225, 287, 339–341
Systematic-, 24

T

Table
Definition-, 443
Error-, 437
GLOSSARY-, 431
Hash-, 215
Keywords-, 52
Literal-, 53, 68
Modelling-, 393
Name-, 443
Regular-, 75
Symbol-, 48, 50, 52, 53, 58–62, 68, 71, 131, 139, 159, 160, 167, 170, 173, 176–178, 180, 210, 213, 215, 313, 315, 317, 348, 361, 362, 434, 436, 439, 450
Void-, 477
Tables
Symbol-, 58
Target
-Back, 228
-Code, 6, 10, 12, 25, 37, 42, 48, 139, 140, 153, 160, 161, 239, 268–271, 273, 275, 289, 291, 300, 310, 320, 327, 328, 348, 349
-Generator, 269
-Index, 237
-Instructions, 269, 282
-Language, 6, 7, 12, 21, 24, 26, 29, 30, 41, 46, 113, 134, 136, 140, 182, 185, 227, 228, 239, 248, 267–270, 275, 276, 283, 348, 349, 353
-Machine, 11, 19, 21, 26, 29, 134, 140, 248, 270, 271, 276
-Machines, 134
-Operating, 287
-Platform, 334
-Processor, 271, 272, 282, 289
-Program, 270
-Seems, 270
Techniques
Compiling-, 165, 331
Proof-, 403
Testing , 384, 462, 463
Theoretical
-Viewpoint, 10
Theory
Language-, 409
Mathematical-, 392
Notes-, 398
Threaded, 208, 227, 244–247, 266, 341
Threaded
-Code, 227, 244–247, 347
Threads, 44, 45, 243
Time
Off-, 321
Provides-, 320
Saves-, 322
Token
Characters-, 50, 75
Int-, 69
Intermediate-, 51
Parser-, 50, 75
Return-, 69
Tool
-Came, 46
-Chain, 353, 390
-First, 73
-Lex, 62
-Yacc, 93
Building-, 112
Debugging-, 220
Development-, 349
Generating-, 62, 112
GNU-, 353, 390
Right-, 64
Tools-, 21, 62, 349
Writing-, 21, 27, 46, 50, 62, 73, 75, 93

Tools
-Came, 46
-Lex, 62
-Yacc, 93
Development-, 349
Generating-, 62, 112
Writing-, 21, 27, 46, 50, 61, 62, 75, 93
Trace
Execution-, 39, 406
Transfer
-Instructions, 354
Data-, 354
Translated, 8, 25, 63, 183, 185, 220, 224, 228, 244, 264, 274, 305, 334
Translation, 1, 2, 6–9, 21, 23, 139, 140, 154, 165, 184, 227, 228, 248, 267, 281, 282, 292, 309, 332, 474
Translator, 2, 3, 6, 7, 21, 22, 24, 26, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 62, 164, 398, 424, 434
Tree
Abstract-, 51
Advanced-, 349
Basic-, 363
Binary-, 58, 146, 169, 170
Black-, 58, 349, 391
Call-, 37, 476
Complex-, 231
Constructor-, 170, 179
Define-, 170, 213
Definition-, 171
Derivation-, 425, 426, 429
Filed-, 179
First-, 262
General-, 58, 146
Int-, 171
Last-, 264
NonBinary-, 170
Parse-, 11, 25, 34, 37, 48, 51, 52, 62, 73, 76, 91, 143, 154, 157, 158, 165, 227, 229, 230, 232, 235, 239, 240, 262, 264, 269, 471–474
Partial-, 89
Retrun-, 171
Return-, 171
RHS-, 297
Rule-, 174
Sample-, 231
Search-, 58, 211, 221, 231, 349, 362, 440
Tree
Semantic-, 153
Single-, 76
Sophisticated-, 349
String-, 433
Struct-, 230
Sub-, 297
Syntax-, 15, 18, 37, 63, 73, 81, 158, 227, 233, 235, 239, 240, 260, 265, 270, 297, 414, 415
Table-, 362
Tested-, 384
Total-, 230
Type-, 169–171, 362
Triple
-Notation, 238
Tuple
Ordered-, 398
Tuples
Ordered-, 398
Type
Additional-, 170, 338
Appropriate-, 175
Argument-, 167
Attribute-, 153
Automatic-, 334
Basic-, 167
Binary-, 170
Boolean-, 141
Chomsky-, 50, 76, 414, 415
Class-, 158
Comparing-, 171
Complex-, 170, 171
Correct-, 159, 166, 167
CPU-, 272
Data-, 59, 62, 121, 153, 166–168, 171, 173, 193, 239, 241, 278, 279, 334, 338, 349, 360, 362, 474, 475, 478
Declared-, 153, 177, 178
Define-, 167, 170, 213
Defined-, 167
Demonstrating-, 349
Enforce-, 167
Error-, 114
Expected-, 134
Explicit-, 159, 173
Field-, 167
Five-, 116
Function-, 177, 178
Functions-, 166
Identifier-, 50, 75
Implicit-, 173, 175
Implicitly-, 173
Including-, 176
Infer-, 172
Input-, 8
Int-, 32, 399
Integer-, 338
Involve-, 173
Machine-, 183
Mentioned-, 174
Named-, 171
Node-, 261
Object-, 134
Omit-, 172, 173
Optimization-, 315
Own-, 167
Permitting-, 172
Pointer-, 134, 172, 260
Preliminary-, 175
Primitive-, 180, 208, 276
Record-, 456
Recursive-, 171, 172
Reduce-, 89
Regarding-, 141
Result-, 176, 177
Return-, 133, 181, 366
Scalar-, 337
Second-, 289
Short-, 32
Special-, 308, 338
Stack-, 257
Statement-, 25, 28
Statically-, 172

Strict-, 166
Strongly-, 166, 242, 334, 335
Structuring-, 170
Sub-, 179, 180, 213, 257, 362
Supply-, 159
Support-, 173
Supported-, 360
Symbol-, 159
Syntactic-, 53, 58, 117, 179, 362
Terminal-, 183
Time-, 141
Token-, 31, 32, 35, 38, 53, 158
Types-, 2, 4, 5, 9, 11, 15, 19, 75, 94, 116, 124, 127, 133, 136, 140, 141, 150, 154, 159, 160, 167, 168, 171, 172, 177, 182, 196, 227, 257, 283, 314, 334, 338, 349, 350, 364, 428, 436, 451, 453
Type
Union-, 124, 153
User-, 220
Uses-, 181
Value-, 153, 176, 179, 242, 267
VAR-, 158, 257
Variable-, 364
Via-, 171
Weakly-, 167
Words-, 166
Types
Additional-, 170, 338
Argument-, 167
Basic-, 167
Complex-, 171
Data-, 59, 153, 166–168, 171, 241, 278, 279, 334, 338, 349, 360, 475
Declared-, 177
Defined-, 167
Enforce-, 167
Expected-, 134
Five-, 116
Identifier-, 50, 75
Infer-, 172
Input-, 8
Named-, 171
Node-, 261
Object-, 134
Primitive-, 180, 208, 276
Recursive-, 171
Regarding-, 141
Statement-, 25
Structuring-, 170
Syntactic-, 53, 179
Token-, 31, 32, 35
Value-, 153, 179, 242

U

Ullman, 137, 165, 181, 225, 331

V

Validate, 61, 139, 181, 295, 301
Value
-Associated, 66
-Comparison, 308
-Numbering, 310, 316, 318
-Numbers, 317
-Pass, 207
-Type, 267
-Types, 124
Valuenumbering, 318
Values
String-, 358
Variable
Checking-, 174
Dead-, 322
Environmental-, 222
Live-, 326
Parameters-, 187
Variables
Environmental-, 222
Parameters-, 187
Versions, 19, 23, 61, 62, 69, 124, 137, 173, 186, 206, 220, 221, 228, 236, 279, 287, 333, 335, 336, 338, 341, 349, 451
Viable
-Prefix, 93, 96
Viewpoint
Equivalence-, 171
Human-, 2
Language-, 71
Optimization-, 291
Set-, 270
Theoretical-, 50, 75
Virtual
-Addresses, 222
-Computer, 333
-Extended, 309
-Machine, 11, 24, 26, 44, 46, 227, 228, 235, 238, 241, 242, 247, 248, 265, 266, 333, 338, 346
-Machines, 227, 235, 248
-Memory, 184, 185, 191, 216, 287, 290, 309, 451, 455, 456, 458, 459
Visibility, 159, 160, 182, 197, 224
Vmgen, 346, 347
Vocabulary , 410, 431

W

WAM, 227, 248, 249
Words
Allocation-, 346
Bit-, 276
Byte-, 358
Defined-, 342
Defining-, 346
Double-, 390
Immediate-, 343, 345
Parsing-, 343, 345
Process-, 333
Reserved-, 47, 48, 73

Y

YACC, 21, 27, 32, 46, 53, 62–64, 69, 73, 75, 80, 93, 103, 112–115, 117–124, 126–128, 134, 136, 137, 139, 150–153, 158–164, 178–180, 229, 249, 252, 349–351, 364, 367–369, 371, 384, 474