Compilers – The Basics

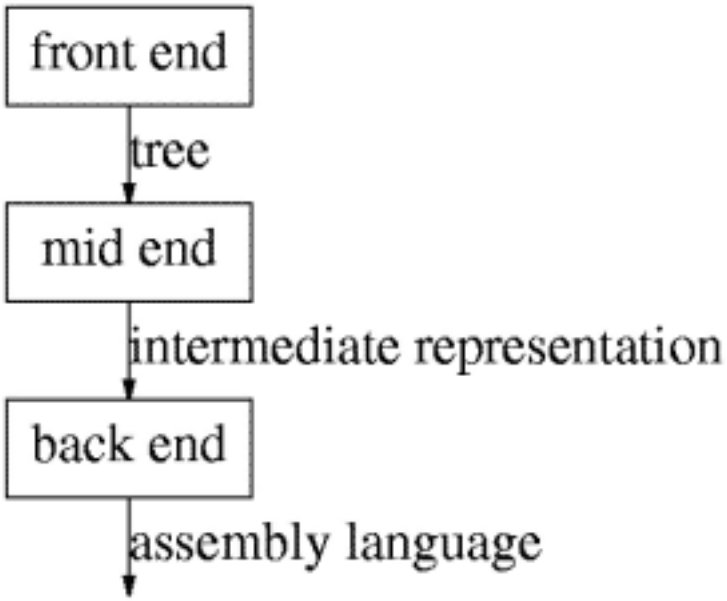
The Compiler

Since the Pentium 4 was launched its poor performance on unoptimized code has triggered interest in compiler technology. This article is an introduction to the form and function of compilers.

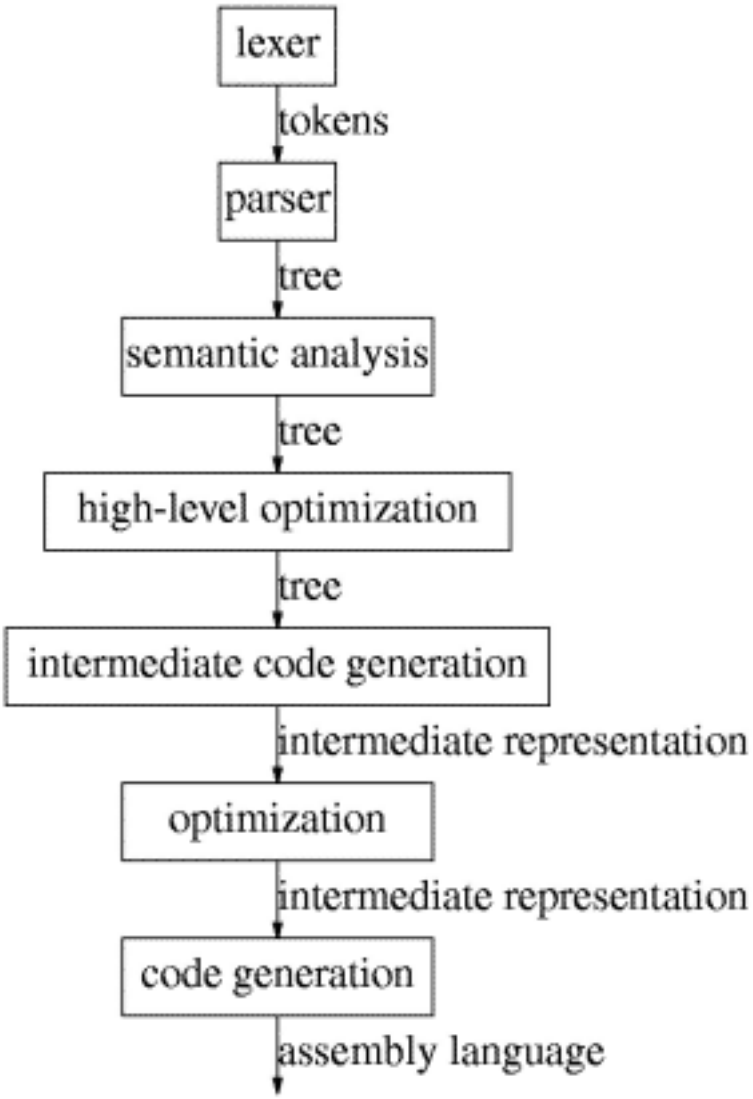
What is a compiler?

Programmers think of a compiler as a program that takes input in the form of a high level language and produces output in the form of assembly language (or machine code) for some processor. This is actually too limited of a definition: a compiler by the purest definition takes a string and outputs another string. This covers all manner of software; text formatters such as TeX and troff convert an input language into a printable output, such as postscript. Programs that convert between file formats or different programming languages are compilers. Many interpreted languages such as Python, Forth and Smalltalk compile internal bytecode and interpret it. Many other programs use compiler like structures to interpret configuration files. Also, pretty printers, indentation & coloring in code editors and static type checkers (e.g. lint) use techniques similar to compilers. Have I mentioned web browsers yet? In this article, however, I will be discussing what is normally regarded as a compiler, as these are the most sophisticated and interesting.

When high level languages were first invented in the forties and fifties no compilers had been written. Early compilers were complex and took huge amounts of time and manpower to write. Since then work on methods and tools have made it possible for a single programmer to write quite an advanced compiler. One of the main lessons learnt is how to split a compiler into parts. At the highest level there are three parts: the front end that understands the syntax of the source language, the mid-end that performs high level optimizations and the back end that produces assembly language.



At a slightly lower level there are seven stages. Each of these stages will be explained here. Not all compilers have all these stages, but most have at least five or six of them. The information is passed between these stages in the data structures marked between the blocks.



The Lexer (or lexical analyzer)

/bb|[^b]{2}/ – Hamlet

The lexer is the first small step in understanding a language. Its purpose is to decompose the stream of input characters into “tokens”. This is best understood by example. For instance in the C language the statement:

	char str[] = “A good line.”;

Decomposes into:

token 1: Keyword, specifically “char”

token 2: Identifier, specifically “str”
token 3: Left square bracket
token 4: Right square bracket
token 5: Equals sign
token 6: String, specifically “A good line.”
token 7: Semicolon
Whole strings are tokens, as are some individual characters, such as ‘=’. Each token is labeled as belonging to a certain type, such as “identifier”. Some have extra information associated with them; for example the string token contains the string in question. For most languages the lexer is simple, though there are exceptions such as Fortran and Perl. Languages exist that can generate lexers from specifications, such as ‘lex’ and ‘flex.’ These languages specify the form of a token using a regular expression.
A regular expression (RE) is a form of search. For instance ‘function’ is a valid regular expression that will search for the word ‘function’. The ‘f’ at the beginning searches for the character ‘f’, if I had written [Hh] it would mean look for ‘H’ in either case. [] defines a group of characters. Another example is [A-Z], which searches for a capital alphabetic letter. Following any such expression I could put ‘+’ meaning one or more. So, for instance [a-z]+ means one or more lower-case letters. A really useful RE is one to find an identifier, how about [A-Za-z_][A-Za-z0-9_]* this means one alphabetic letter followed by zero or more alphabetic letters or digits. [^b] means any letter but b (which may help you decipher the quote above), there are more symbols that can be used in REs I won‘t describe them because they are made to save typing, like [^b]. `lex’ and `flex’ are fed a list of regular expressions each with an associated action, which passes the token to the next stage, the parser.
Users of *nix systems will know all about REs since they are used for searching and replacing in many editors, utilities and text processing languages for similar purposes. Regular expressions were not always for processing text. They invented by S.C. Kleene in a paper called “Representing events in Nerve Nets” in 1956 as an attempt to understand nerve activity based on the flawed McCulloch-Pitts neuron model. Since then they have also been used in speech recognition and to detect flaws in printed circuit boards. Today they are heavily used in DNA sequencing. For an explanation of how they work see http://lambda.uta.edu/cse5317/notes/node1.html.

The Parser

“Top reason why compilers are like women: Miss a period and they go crazy” – anon.
Parsing is the process of understanding the syntax of a language, to allow it to be represented by the compilers internal data structures. The most sophisticated ideas humans can relay to computers are communicated with programming languages. This has made programming languages compromises between the human thought process, the computer’s execution process and the

computer's capability to understand language. The parser deals with the last facet of the problem. As such a lot of research has gone into this area.

The are many ways of automatically recognizing language. In the field of programming languages there are only two significant methods: Top-down parsing and Bottom-up parsing. Top-down parsing partitions a program top-down, programs into modules, modules into subroutines, subroutines into blocks. Bottom-up techniques group tokens together into terms, then expressions, statements, then blocks and subroutines. The distinction will become clear as they are explained. Before explaining how a language is recognized it is useful to explain how its rules, its grammar, can be expressed.

Grammar

The grammar of a language defines what comprises a meaningful statement of the language. It doesn't deal with what it means, but that can be added as extra information. Grammars are specified using "productions." Here is an example of a production:

statement -> if (expression) statement else statement

What does it mean? The part to the right of the "->" defines a phrase in a language. The part on the left is the class the phrase falls into. This production can be read "a kind of statement is the token 'if' followed by an expression in brackets followed by a statement, then the word 'else', ending in a statement." There will be further rules defining what a statement is, such as:

statement -> while (expression) statement

By collecting many productions a grammar is formed. Many such grammar specifications are available on the net (ftp://ftp.iecc.com/pub/file has C, C++, Java & Delphi grammars). Notice that with production, more complex language can be described than with regular expressions. In particular, productions are recursive, so in the above a statement can be defined to include statements, which isn't possible with regular expressions. Productions were invented by Emil Post and first used to specify a programming language (Fortran) by John Backus (hence Backus-Naur form). It is interesting to note that they were also invented between 200BC-400BC by Panini to specify the rules of Sanskrit grammar.

The Parser (cont.)

Top-down Parsing

A common top-down parsing technique is "predictive parsing". It can be used quite easily to parse C and Pascal. The more general method of "recursive descent" with backtracking can deal with C++. Predictive parsing is the most direct and elegant parsing method, and many languages, such as C, were developed with it in mind. To understand how this works I'll explain how it is applied to a simple grammar.

expression -> term

		 | expression + term

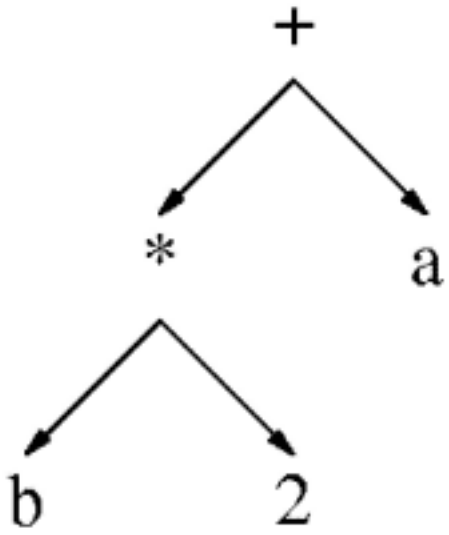
		 | expression – term

This is shorthand for the three productions:
expression -> term
expression -> expression + term
expression -> expression – term
also:
term -> number
	 | term * number
	 | term / number
Number can be defined by a regular expression (e.g. [9-0]+). It is a token.
This grammar understands simple arithmetic. Here I have called the component parts of the language expressions and terms, but I could have called them anything. To make a predictive parser, a subroutine ‘expression’ is written to parse expressions. This subroutine immediately calls ‘term’, a subroutine to understand terms. ‘Term’ then searches for a number token. After this it searches for a * or /. If it finds one it looks for another number. If it doesn’t find * or / its job is over and it hands back what it has found to ‘expression’, which attempts to finish the job.
This is the general pattern of a predictive parser. Go to the subroutine that understands the most basic phrase, when it can’t understand what it is given go to a higher level subroutine, and so on. In this way a top-down parser fits in nicely with the concept of top-down design, since the highest level subroutines call lower level routines to do their work for them. It is called a “predictive parser” because the higher level routine has to predict which of several lower level routines to call based only on the first token of the phrase that one of the routines must understand. The first token must predict the subroutine call. This is a serious limitation, but irrelevant if the language has been designed with predictive parsing in mind.
The parsing subroutines in the above example can do other things than recognize tokens; they can perform actions on the way. For instance when a term is recognized by the rule:
term -> term * number
The calculation of term * number can be performed and the result returned to the calling subroutine, expression. This could be written:
term -> term * number { return term * number }
similarly
expression -> expression + term { return expression + term }
Now you can see why I divided the arithmetic problem into terms and expressions. The terms are evaluated first, then their results are passed to expression, so the rules of “precedence” in arithmetic are treated correctly. Multiplication and division come before addition and subtraction. This is how precedence is dealt with in a recursive descent parser. In a compiler, the actions associated with each production could be used to generate assembly language, but generally they will be used to generate a data structure called an abstract syntax tree I alluded to earlier.
Bottom-up Parsing

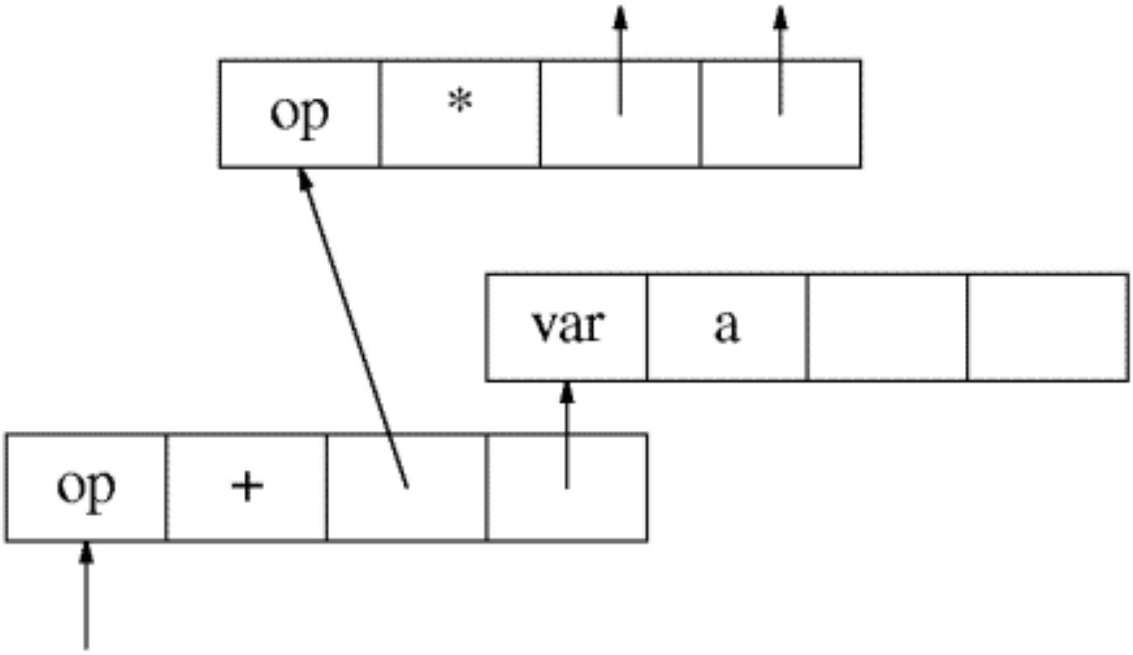
In contrast to top-down parsing, bottom-up parsers are almost never written by hand, they are generated directly from grammars using parser generators such as Yacc and Bison. In fact, they are quite difficult to write, but once a parser generator is written that problem is solved. Yacc is an acronym for “Yet another compiler compiler”, while Bison is an extended version of Yacc. A bottom up parser starts by grouping together all the tokens on the right hand side of a production. When faced with:
8 + 7 / 5
it will group “7 / 5” then group that with “8 +”. This approach isn’t limited by the same problems as the predictive parser. Situations where the first token of a production doesn’t indicate how to parse it aren’t a problem. Bottom-up parsing has other problems of its own, specifically it isn’t always clear whether to add some more tokens to a group or start applying a new production. E.g.
statement -> if (expression) statement else statement
		 | if (expression) statement
In this case when the parser has seen “if (expression) statement” it doesn’t know whether to look for “else statement” or look for the next statement. This is a “shift-reduce” conflict. When specifying grammars in Yacc or Bison there are ways of eliminating these conflicts. Bottom up parsers also tend to have more problems with errors than top-down ones. They understand all the individual bits of a phrase before the highest level part, by which time the remaining bits make no sense. Although I haven’t mentioned it, when grammars are created for languages, generally more productions are added to deal with errors. e.g.
statement -> if !(
(where ! means not)
Will catch ‘if’ statements not followed by a bracket. The compiler can then give a useful error message such as “expression in if statement must be in brackets on line X”. This kind of stuff makes the grammar much more complex, but it is worth it to give sensible error messages.
For many languages bottom-up parsers are the only choice, because the languages have grown around a parser generator. I have often thought that the way humans understand language is a mixture of bottom-up and top-down methods. Particularly, human languages suffer from many shift-reduce conflicts, for example adding ‘not’ to the end of a sentence reverses its meaning (think “Wayne’s world”). Often, clever writing and script-writing depends on changing the meaning of words at the last minute it is possible to do so, relying on the readers mind to have already drawn the conclusion.

The Tree

Tree (actually abstract syntax tree) is the data structure used to represent the meaning of the code in the next few stages of the compiler. When a parser matches the left-hand side of a production it adds something to the tree. Here is a little bit of tree.



This tree is created in memory. Every part of the left-hand side of a production produces a pointer. The right hand side gets some memory, stores the instruction, instruction type (eg. operation, variable, function call), and the pointers it recieves. It then passes a pointer to this structure to its left-hand side. In the predictive parser this roughly means that each subroutine gets pointers from every subroutine it calls, collects them in memory, adds the instructions to be performed and returns a pointer to them.



Trees can be easily written using brackets. Each left bracket represents a new node and each right bracket the end of that node. The tree above could be written as (+ a (* b 2)). This implies that the entire process of parsing can be omitted and all that is needed is a language where every concept is de-marked with brackets. A language has been written like this: Lisp. Lisp is an acronym for LiSt Processing Language, but it is more accurately called TRee Processing language. However, John McCarthy invented in the 50's where he would never have got away with calling it TRIP. In Lisp, linked-lists are a fundamental data type, which means that data has the same form as code. In fact in Lisp there is a function call `eval' that passes a list to the interpreter for execution, so it is possible to construct small sections of code and execute them, or allow the user to input code. It also makes it easy to recognize or generate Lisp.Once the tree is generated it is then 'decorated' or 'annotated'. This means adding type information to it and quite possibly other information useful to optimization. This stage finds type mismatches. Some languages stop at this stage and interpret the resulting tree without compiling. Perl does this as do some lisp interpreters. Also, languages that generate images rather than code, such as HTML and XML, sometimes build those images from trees.
A common Linux kernel problem is: "I keep getting internal compiler errors when building the kernel with make -j". The reason for this, as every F.A.Q. replies, is that system memory is deliberately overclocked or of low-quality, and this is the most common symptom. This demonstrates something of the nature of compiling; huge complex data structures being read, annotated and replaced. Also several simultaneously running compilers is a good test of system memory.

Glue

I've quietly skipped over some quite important details. Firstly…

The Symbol table

Every subroutine name, variable name, constant and type must be recorded along with information about it. For instance, variable names need the types associated with them, a storage location in memory at runtime and an associated scope. Subroutine names need locations and argument lists annotated with types. This is done by using a "hash map", which is a table that associates a string (e.g. an identifier) with a structure or list containing this information. Scopes must be taken into account. Finally, if the user requests, all the symbol table data is kept until the code is generated and then annotated to the executable for use by a debugger.

Error detection

Error detection is an ongoing process occurring throughout the front end. The lexer can detect malformed tokens, the parser can detect syntax errors and the tree can detect annotation type mismatches. The parser should be able to 'get past' at least some errors so the user has useful error messages from a long compile.

Next time

The next part of this article will deal with optimization and code generation.

Useful references

http://www.iecc.com – Several resources and archive of the comp.compilers newsgroup.

http://compilers.iecc.com/crenshaw – A tutorial creating a one-pass compiler for a superset of Pascal in Pascal. Free, excellent explanations. Uses a predictive parser and no tree structure.

Computer architecture and Organization: A Quantitative approach, J. Hennessey and D. Patterson. – The bible of computer architecture, contains much material on loop optimizations. Compares compiler methods to hardware methods.

Compilers: Principles, Techniques and Tools, A. Aho, R. Sethi J. Ullman – `The Dragon book'. The bible of compilers, written in 1986 this books information on front end aspects is still solid. Relatively little information on optimization (still more than many compiler books).

http://lambda.uta.edu/cse5317/notes/node1.htm – University of Texas course notes on Compilers by Leonidas Fegaras

This is the second segment of a three part series exploring the various aspects of compiler technology.

Code optimization is today the core problem of compiler design. Although some work still goes into parsing methods, most research targets the problems of optimization. There are broadly two types of optimizations, optimizations for speed and for memory space. Sometimes an optimization does both. Before I go much further it is worth mentioning what optimizations provide.

Why Optimize?

Memory

First, memory space. How much code does it take to fill up the memory of a modern PC? A lot is the answer. Modern operating systems take multiple megabytes of disk space, but much of it is graphics, sound and documentation. Also virtual memory means that large executables don't block up memory much. For most practical purposes size doesn't matter anymore. All that matters is the consumption of instruction cache, which can affect speed.

The exception is the embedded world where bytes still matter. Embedded system developers have resurrected methods from decades ago to face this. Recently they have also developed optimizations especially targeted to save power.

Speed

There are two types of programs those that need to be fast and those that don't. With ever increasing speed of processors and peripherals more applications fall into the latter category. Of those where speed is critical there are several possible bottlenecks:

- Hard disk & File system
- Network
- Operating system kernel
- Languages standard library
- Users
- Memory
- Processor

The compiler can affect only the last strongly. Sometimes it can improve memory bandwidth or latency. Operating system and language libraries maybe affected by the compiler they are compiled with.

Together this means that many if not most programs don't benefit from optimization.

Endless Possibilities

A vast range of optimizations has been applied and studied, here I will concentrate on the most important ones, the most important thirty odd that is. It is difficult to classify optimizations, crudely there are two types:

1. Elimination of unnecessary work (henceforth "strength reduction").
2. Selection of the best patterns of instructions.

These two overlap each other at the edges since selection of the best pattern of instructions involves preventing the processor doing unnecessary work. Below is a list of the various types of

optimizations; the higher level optimizations will be first, with the lowest level of optimizations being covered in the last part of this series.

1. Strength reductions
- Dead code elimination
- Hoisting of invariants from loops
- Common sub-expression elimination
- Jump removal
- Constant folding
- Inlining
- Trace scheduling
- Arithmetic simplification
- Tail call optimization
- Iteration order reversal
- Loop unrolling
- Induction variable elimination

2. Selection of efficient instruction patterns
- Loop fusion
- Loop fission
- Prefetching
- Blocking
- Memory bank optimization
- Instruction combination
- Instruction selection (peephole optimization)
- If conversion
- Register allocation
- Register movement elimination
- Instruction scheduling
- Delayed branch scheduling
- Vectorization
- Threading

Some of these optimizations are done when the program is represented close to its source form, as for example tree, others are done later when it is in a low-level form. This lower level form is close to assembly language in semantics. In general each optimization is performed one at a time. A subroutine applying the optimization goes through the tree, does some stuff and then another subroutine does more stuff, etc. These are passes. The separation of the optimization problem into multiple passes makes developing or altering several optimizing passes at once possible. Of course it doesn't have to be done this way, but this is how it is almost always done, see [2] for one alternative.

I'll introduce one further piece of jargon: the `basic block' is a section of code with no jumps, into or out of it. It's a useful idea because doing optimizations within basic blocks is much easier than doing them everywhere.

I apologize if some of this isn't particularly interesting; optimizations are determined by necessity, necessity is a horribly uneven writer, but when it's good no one can touch it.

High-level optimizations I

Dead code elimination

Programmers occasionally write code that can't be reached by any possible path of execution. For instance checking 'a = 1' then changing it and accidentally checking if 'a = 1' again. This is dead, unreachable code, it is actually quite uncommon and rarely affects performance or code size. It is found mainly to provide warnings to programmer because it is sometimes associated with bugs. Also code generated automatically often has unreachable parts.

Arithmetic simplification

Programmers generally give algebraic expressions in their simplest form, but not always, simplification deals with this. A set of algorithms applies arithmetic identities attempting to find the one with the smallest set of instruction. This is not simple, doing it generally is not possible. For e.g. would you know that:

```
sqrt(exp(alog(x)/y)) = exp(alog(x)/2*y)
```

Your compiler probably will, but only because the calculation occurs in the whetstone benchmark, so compiler designers have tuned for it.

Constant Folding

Find out at compile time that an expression is a constant for example if you write 2 * PI * x / 4 it will reduce to 1.570796337 * x. This is very useful, especially because every compiler for many years has performed it so programmers know it happens and rely on it, making programs clearer.

Common sub-expression elimination (CSE)

Programmers often repeat equations or parts of equations, e.g.

```
x = sqrt(M) * cos(&#952;);
y = sqrt(M) * sin(&#952;);
```

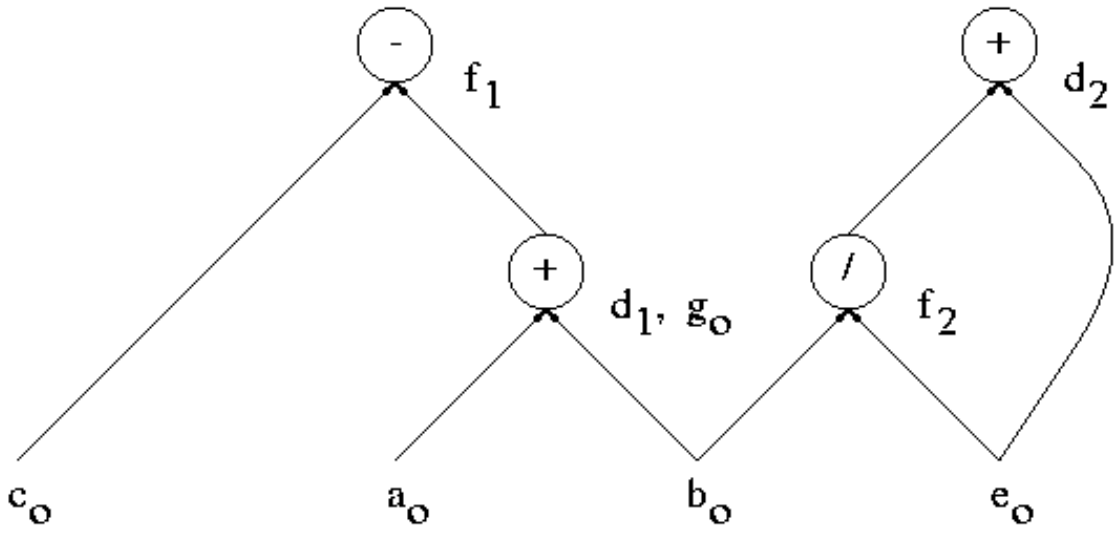
In this case the common subexpression is the "sqrt(M)" part, the CSE stage will store the result of the first sqrt(M) and reuse it instead of recalculating it. Unlike the other two optimizations I have mentioned there is a problem with CSE, it can make a program slower in certain circumstances. In particular when eliminating the duplicated code its result must be stored so it can be reused. In some cases (if it is stored in memory) loading and restoring from memory can take more time than redoing an operation. Problems like this must be accounted for. In general most optimizations run the risk of making things worse. Common subexpression elimination is very important in array address calculations since some parts of the calculations can be reused. You may have heard of GCSE; Global CSE is elimination across basic blocks.

At this point is worth saying a little about the word `global', which unfortunately has confused meaning. When people speak about `global-optimizations' they usually mean optimizations of whole programs, this is similar to the idea of global variables. `Global-CSE' though merely means `over many basic blocks', as does global register allocation.

Data flow

Most optimizations deal with data flow, so code is brought into that form: a directed acyclic graph. Here is the dag of some expressions:-

```
d = a + b
f = c - d
f = e / b
g = b + a
d = f + e
```



a + b is a common to d = a + b and g = a + b, so instead of creating a new node for it, the node is reused, so common subexpressions are found. Here the code building the dag knows that + is commutative, it may also apply other algebraic rules. f in f = c – d is overwritten by a later assignment and is therefore dead code. So dead code can be found by looking for nodes with no variables left alive.

High-level optimizations II

Copy propagation

Deals with copies to temporary variables, a = b. Compilers generate lots of copies themselves in intermediate form. Copy propagation is the process of removing them and replacing them with references to the original. It often reveals dead-code.

Inlining

Inlining is a conceptually simple optimization. When a short subroutine call is found it is replaced by the contents of the subroutine. This is a complex optimization to perform for several reasons. Firstly the same function must be placed in different subroutines; different variables are assigned to

registers in each subroutine. This means the only way to efficiently inline subroutines is to expand them in the context they appear each time they appear (using different registers for example). The second related problem is one of a group of “phase ordering problems”; doing one optimization before another sometimes has the combined effect of producing inferior code. Specifically code should be inlined early in the compilation process to allow further passes to perform optimizations on the inlined version. Whether inlining is a good idea depends on the final length of the inlined version, which is not known until close to the end of the process. There are several partial solutions to this problem:

1. Advance compilation through the other passes and find out how much code is generated.
2. Estimate from available information how much code will be generated.
3. Use several inlining passes at different stages of compilation.

Using these methods reasonable inlining can be performed most of the time, at the cost of quite a lot of complexity. Note also that in C where all functions are global and may be declared in another file a non-inlined version is always needed in case this happens. It maybe subsequently removed by the linker if no such declarations occur.

The form of generated code

The forms of compiler generated code vary, but most are similar. Specifically they:

- Have a small number of registers for commonly used values.
- Perform arithmetic operations (i.e. +,-,*,/) on integers and floating point numbers.
- Apply Boolean operations, such as `or’ or `and’ to registers.
- Load and store data to addresses in memory.
- Jump to a piece of code at some address in memory.
- Make decisions, normally by jumping or not jumping on the basis of the value of a register or flag.

As far as I am aware all modern compilers assume roughly this model of a processor throughout most of their code. Extra instructions the architecture supports are either ignored, or used only in libraries or used by grouping simpler instructions prior to generating assembly language.

In order to store global variables compilers simply use static memory space. To store the return addresses of subroutines modern compilers use a stack. A stack is like one of those spring-loaded plate-warmers, if you’re reading this you probably know what I mean. Local variables, parameters to be passed, temporaries that won’t fit in registers and return values are also stored on the stack. At the beginning of a subroutine code is inserted to allocate space for them, the `activation record’. At least the parameter passing part of the activation record must be standard, if it wasn’t object code and libraries compiled with different compilers wouldn’t be compatible. Compilers utilize the registers to store values, these registers must be saved somehow. So they are put on the stack, generally there are two methods. Firstly, either the function being called saves them by storing them on it’s stack and restoring them before returning: `callee save’. Or the function doing the calling saves them, `caller save’. There are trade-offs with both. Doing too much work in callee’s makes inefficient the lowest-level functions, but there are a lot of callers too. The answer is a compromise, a group of registers are set asides as caller saved, if a subroutine is short enough to get away with using only those it need do no saving itself. The rest are callee saved, this normally includes infrequently used

registers like floating point and vector registers. Code to do the saving and restoring at the beginning and end of subroutines is called the prologue and epilogue respectively. All this stuff must be specified to allow code to inter-operate this is an ABI – Application Binary Interface specification. Whole program optimization can improve this further. The `linker' is normally just a means of joining object files together to form an executable. Whole program optimization involves burying a whole or partial compiler in it. This allows all the calls internal to a program to be optimized in anyway the compiler sees fit. To make this possible object files are sometimes not object files at all, but a binary representation of the compilers internal representation.

High-level optimizations III

Tail call optimizations

		`*To iterate is human, to recurse divine' – L. Peter Deutsch*

As I said earlier subroutines are dealt with by pushing the return address onto the stack and popping it off at the end of the subroutine. This creates an interesting possibility. Consider the code:-

```
sub fred {
   jim;
   sheila;
   }
```

`jim' and `sheila' are calls to other subroutines. Also `sheila' is the last piece of code before the end of the subroutine, a `tail call'. Normally the address of the instruction after `sheila' (the first instruction of the epilogue) would be placed on the stack then sheila would be called. As sheila finishes it will pop the return address of the stack and use it, now returning to the end of `fred', the address fred was given will be popped off the stack. A trick can be done here, sheila can be jumped to without recording anything on the stack, if this is done the return by popping the stack will still work right. At the end of sheila the return address of fred will be at the top of the stack.

Okay what's the point? Calls aren't that expensive. A `self-recursive call' is a subroutine calling itself. A `self tail call' is a recursive call and the last call in a routine. A special optimization can be implemented in this case (which isn't that rare), the call can be replaced by a straight jump. Because the call is executed many times in this situation it is a profitable trick. This is an interesting optimization because it has effects that ripple far from the reduced call overhead. If any arguments are passed then once optimized the activation record isn't rebuilt each time. Imagine a recursive routine that is called 1 billion times and has one integer in its activation record, it'll use over 4GB of memory, whereas after the optimization has been done it will use 4 bytes. So the routine will crash or seg-fault without the optimization and work with it. As you may imagine this can't be used in portable code. Lisps have had this optimization for decades and lisp programmers often rely on it, a very useful state of affairs.

e.g.

```
foo()
{
```

```
    if (l = "X") {
       bar();
       foo();
    }
}
```

becomes

```
foo()
{
    L: if (l = "X") {
       bar();
       goto L;
    }
}
```

Loop optimizations

Very commonly most of the work of a program is done in one or two small loops that are iterated huge numbers of times. The following high-level optimizations are targeted specifically at loops because of the large amount of work they do. In mathematical code the situation I mention above where most of the work is done by a loop is almost universal, for this reason loop optimizations are the most important. Compilers for supercomputers and number-crunching clusters spend most of their time and have most of their complexity in performing loop optimizations. Loops often access arrays and many loop optimizations rely on this fact. Programmers know the importance of loops and only pay attention only to their code when evaluating algorithms.

Loop invariant hoisting

This optimization is often referred to as “strength reduction” although properly strength reduction is a more general term. Anyway, the operations in the body of a loop are done many more times than those outside the loop, so it’s better to do as much as possible outside the loop. Loop invariant hoisting moves loop invariant calculations out of the loop. Advanced implementations work out whether it is better (or possible) to move the invariant part out to the beginning of the loop or the end.

High-level optimizations IV

Loop unrolling

The motivation for this optimization is the same as invariant hoisting; code in the middle of a loop is executed many times so more resources should be spent optimizing it. Unrolling means copying out the body of the loop several times and changing the loop index correspondingly. Doing this means that the time spent executing the branch instruction at the end of the loop and updating the index is halved. This may appear a trivial gain but if the loop body is small it can be very significant indeed. For instance the following assembly language adds a value in floating point register F2 to every element of an array:

```
Loop:
   LD   F0, 0(R1) ; get array element no R1 and store in F0
```

```
      ADDD F4, F0, F2 ; perform add
      SD   0(R1), F4 ; store result back where it came
      SUBI R1, R1, #8 ; R1 = R1 - 8; 8 bytes = 1 double precision floating
point number
      BNEZ R1, Loop ; Branch if not equal to zero
```

This example is drawn from [1]. Using the very crude assumption that the processor always executes an instruction per clock cycle in the above two cycles per loop iteration are spent performing the subtract and the branch instructions. Unrolling the body of the loop once more gives:

```
   Loop:
      LD   F0, 0(R1)
      ADDD F4, F0, F2
      SD   0(R1), F4
      LD   F6, -8(R1)
      ADDD F8, F6, F2
      SD   -8(R1), F8
      SUBI R1, R1, #16
      BNEZ R1, Loop
```

Under the crude assumption (one instruction per cycle) I gave before this loop will execute in 80% the time of the original version. This maybe taken further, it could be unrolled 3 or more times. Each unrolling will reduce the cost of the SUBI & BNEZ instructions, the “loop overhead”. There is of course a limit where further unrolling gains little benefit; but it varies from machine to machine and loop to loop.

Modern microprocessors gain much more from loop unrolling than indicated above for several reasons. Most importantly they are heavily pipelined this means that branches are expensive operations. Processors have branch target buffers(BTBs) and branch predictors once the loop has iterated once the target address of the branch will be stored in the BTB. However, in some processors the BTB and branch predictors take considerable time to produce addresses, slowing the loop down considerably. More importantly on in-order processors loop unrolling gives a latter optimization namely scheduling the ability to make more parallel code. Scheduling reorders the instructions to reveal parallelism. The example above will be extended in the discussion of scheduling later.

Loop unrolling has a number of problems; first the compiler has no real way of knowing which loops in a program do the work. This means it must guess, which is quite tricky, although estimates can be made from the number of iterations to be performed and the number of nested loops. It is only really worth unrolling loops that have a small amount of code in the body, since in larger ones the loop overhead will be small. After all unlikely candidates are removed two things should have happened: First, the loops that do the work should be unrolled. Secondly, few loops should be unnecessarily unrolled. If they are unrolled the code grows in size considerably and sometimes runs slower. Sometimes the range over which a loop is executed is fixed, e.g. 0->100. More often it is variable, if so another loop must follow the unrolled loop to take account of the possibility that the range ends on an odd number (or more generally a number not divisible by the number of unrolls).

Loop fusion and Loop fission

Programmers don’t always split work across loops in an optimal way. For instance sometimes sets of loops that iterate across the same range of the same data, in this case the loops can be fused into one, removing some loop overhead and improving cache behavior.

```
/* Before */
   for (i = 0; i &lt; M; i = i + 1) a[i] = b[i] / c[i];
   for (i = 0; i &lt; M; i = i + 1) d[i] = a[i] + c[i];
/* After */
   for (i = 0; i &lt; M; i = i + 1) {
      a[i] = b[i] / c[i];
      d[i] = a[i] + c[i];
   }
```

More obscure is loop fission. Most loop optimizations rely on the code to have no dependences. A dependence (a “loop-carried” dependence) occurs when one iteration of the loop requires the previous iteration to have occurred before it. This could happen if for instance a statement refers to an array element calculated in the previous iteration. This kind of thing kills vectorization, can prevent scheduling working well after loop unrolling and sometimes prevents loop unrolling altogether. Loop fission is a partial solution if it is possible the loop is split into two loops, one containing the dependence and the other containing the rest of the work of the loop, this second loop can be optimized well.

Induction variable elimination

Sometimes the loop variable isn’t needed, for instance in the following loop

```
   for(i = 0; i &lt; M; i = i + 1) a[i] = 0;
```

i isn’t actually needed, it can be implemented as:

```
   Loop:
      SD   (R1), 0 ; store zero
      ADDI R1, R1, 4
      BNEZ R1
```

Notice this loop also goes backwards. Bringing me to:

Iteration order reversal

Doing a loop in the order the program indicates isn’t always a good idea. In particular, iterating from a low to a high number requires a comparison at the end of each iteration of the loop. If the loop ends on a zero things are easier, no comparison is needed since most machines have a “branch if zero” instruction.

High-level optimizations V

Vectorization

In the seventies Seymour Cray started the supercomputer business around vector processors, processors that can perform floating point operations on large vectors. At this time the work of Kuck on automatically vectorizing compilers was key. Today x86 processors by AMD and Intel have vector instruction set extensions such as SSE. Essentially the same analysis is done as for unrolling. The

scalar instructions are replaced with vector instructions acting on multiple elements simultaneously. In essence if there are no loop carried dependences it can be done, just as unrolling. The stumbling block here is the placement of data in memory. Vector load operations must be able to construct the vector register contents in a small enough time to make it worthwhile, which is tricky. The situation in vector supercomputers is more complex and rather obscure, I won't bother with it.

Memory access optimizations

There are actually several dozen memory access optimizations; most are relevant primarily to clusters. I'll describe a few of the general ones. See "Memory heirarchy design" in [1] for more on these.

Loop interchange

This is primarily a way of optimizing idiotic code. As there are relatively few idiotic programmers out there it is quite uncommon. Most programming languages have a rule that specifies how multi-dimensional arrays are laid out in memory. Programmers who do numerical work in C generally know that a multi-dimensional array is an array of arrays. So the elements accessed by the rightmost array index are adjacent in memory, this is "row-major form". In Fortran the reverse rule is applied, the leftmost index refers to adjacent elements, this is "column major form". When applying an operation to an entire multi-dimensional array it makes sense to iterate through adjacent elements, to do this you must know the rule the language uses. Doing it wrong tends to fill the cache with junk on modern processors. Loop order reversal has the ability to reorder loops to make up for programmer ignorance.

Blocking

Blocking is another multi-dimensional array optimization. The compiler chooses a block size. Instead of going across each whole array it splits up arrays into small blocks that can fit in the cache. Sometimes it is possible to fit the block into registers, for instance on IA64 where there are lots of registers.

Often I here it said that an optimizing compiler or an optimization switch can "buy you 5%". This is totally incorrect, in some circumstances optimizations produce very little, in others they can occasionally produce improvements of more than an order of magnitude. This is especially true of blocking and of the loop optimizations mentioned here.

Memory bank optimizations

Some programmers believe that since computers are binary animals that using power of two to do everything is intrinsically A Good Thing. Sometimes in for instance fast-fourier transforms and radix sorts they are right. This instinct often extends to preferring array sizes that are factors of two. This is often a very bad idea.

Modern memory systems are often capable of servicing multiple requests. Also memory is divided into banks, the address space is constructed by interleaving banks together. This means that when adjacent addresses are accessed, as in iteration through one-dimensional arrays the requests go to different banks. Each bank services them, producing good utilization. The number of banks is almost always a power of two. This means that if a multi-dimensional array is a power of two in size and its

columns are accessed they will be in the same bank. For instance iterating over every element of the array may do this. Since that one bank has limited bandwidth this causes a large slowdown. It can also happen with arrays of structs when the struct is a power of two in size. Having the compiler make arrays that are powers of two in size just a bit bigger can prevent this. Loop interchange can also sometimes help. It is also possible for hardware to solve the problem by mixing up data in the banks, using a method derived from the Chinese remainder theory this is described in [1].

Prefetching

The purpose of prefetching is to bring data that will be needed by the processor in the near future into the processor beforehand. There are two types of prefetching, software and hardware.

Hardware prefetching involves no compiler support, some processor logic predicts the sequence of memory reads the processor will give and brings data in correspondingly. This is only really of use in loops where the sequence is predictable by simple means.

Software prefetching is a method of software-hardware cooperation. Recently it has been introduced in Intel and AMD microprocessors (though workstation RISC processors have had it for years). A new instruction, the prefetch is introduced in general this instruction takes an address as an argument and fills a cache line with the contents of that address (sometimes it uses a register). This makes the data available immediately when it is needed. Like hardware prefetching compiler controlled prefetching only really helps in loops. Prefetch instruction are generally “non-faulting”, this means that if the address they give isn’t in the current processes address space they do not raise an exception. This is very useful behavior from the compilers point of view since it means it can prefetch the next element of the array and not have to worry about running past the end. As with everything there are tradeoffs, a prefetch is an instruction and must be decoded, the compiler must be careful this use of decoding resources does not eliminate any benefit.

References

[1] Hennessy J., Patterson D., “Computer Architecture: A Quantitative Approach”, Morgan Kauffman, 2002.

[2] Krauss http://www.rebelution.net/upper/archive.shtml