Introduction to Compilers

Read Chapter 1 of the Dragon Book. You can also read Chapter 1 of the Mogensen book, but that chapter is very short.

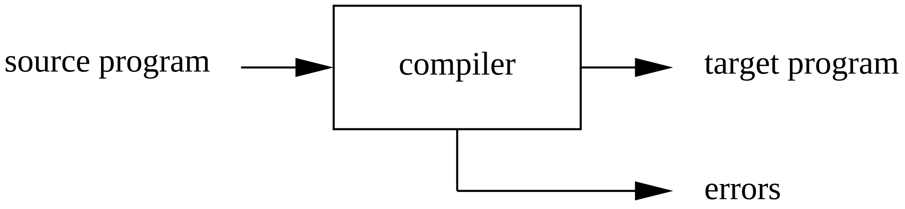
Introduction

This course is about compilers: what they are and how they work, both in theory and in practice. There are several reasons why you want to study compilers.

- You are or will soon be creating software (for your senior project, at the very least, and maybe as a career later) so you need to understand how the tools used for such a task actually work. It is possible to write software without knowing how compilers work but such knowledge will make you a better programmer.
- Many of the things about compilers that you will learn in this course are also relevant to other kinds of software: regular expressions, deterministic and non-deterministic automata, lexers and parsers are all used in many different kinds of applications so this course will also help you use other kinds of software in addition to compilers.
- If you work as a software engineer you will sooner or later have to create or to process data that follows some data language specification (anything from reading configuration files for your software to processing HTML or XML, etc.). Your program will then need ways to read and manipulate that data, which is what the first phases of a compiler have to do too.
- One day you might be lucky and have the opportunity to create your own programming language! You will then have to write a compiler or an interpreter for it (this could be a great senior project too…)
- Compilers are fun! We are going to study several techniques and algorithms in this course which are interesting in their own right.

So let's start with the fun!

A compiler is a program that reads a program written in one programming language (the *source* language) and *translates* it into an *equivalent* program in another programming language (the *target* language). An important part of the translation process is that the compiler should report to the user the presence of errors in the source program, if any. The compiler must also ensure that the target program has the exact same behavior (semantics) as the source program. From the Dragon Book, page 1:

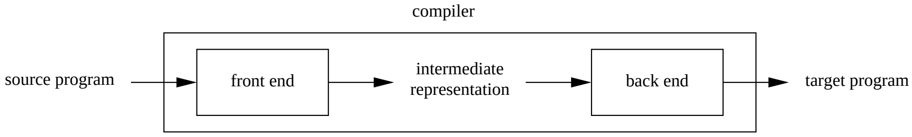


The source language can be either a general purpose programming language, like C or Java, or it can be a domain-specific language, like the LaTeX language I use to create these lecture notes. The target language can be either some other general purpose programming language (as in the case of a C++ to C compiler, for example, which is how the first version of C++ was implemented), or the machine language understood by the microprocessor of a computer (this is the most common case for a compiler), or another domain-specific language, like the PostScript language that is understood by printers and which I use to print these lecture notes.

Compilers vary greatly in complexity. Some are *single-pass* (the program in the target language is generated directly as the source program is read, one expression or statement at a time), some are *multi-pass* with many optimizations, etc. The first Fortran compiler took 18 man-years to create, and it was not a very complicated compiler by today's standards. Some compilers also support several source languages (GCC, the GNU compiler, supports C, C++, Fortran, Java, and Ada) and several target languages (GCC can generate code not only for Intel microprocessors, but also for other microprocessors like the PowerPC, the Sparc, the Alpha, etc., and on several operating systems).

Creating a simple compiler is not as complex as it seems though, for two reasons. First, compilers can be organized in separate phases which can be developed fairly independently of each other, thereby reducing the complexity of the compiler. Second, there are many tools available today to automate the implementation of some of a compiler's phases (most notably lexical analysis and syntax analysis) which greatly decreases the amount of work necessary to get a working compiler. You will be using such tools during the programming assignments.

At a high level, compilation can be broken into two phases: *analysis* and *synthesis*. Analysis breaks a source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. The analysis part of the compiler is often called the *front end* and the synthesis part is often called the *back end*.



Source Code Analysis

Analysis plays an important role in compilers, but also in other kinds of software, so what you will learn in this course about analysis will also apply to other domains:

- Structure editors: text editors that analyze the code of a program as you type it to check that the structure of the code is correct, to allow the user to jump between matching parenthesis in the code, to highlight the code with colors, to detect some errors in your code as you type (like in Eclipse), etc.

- Pretty printers: programs that create a well formatted version of your code, with correct indentation, and different fonts for different parts of the program (a different font for comments, for example, or a different color for variable names).

- Static checkers: programs that try to find bugs in your source program by just looking at it, without running it. Static checkers usually implement very strong analyses (such as very precise type systems, dataflow analyses, etc.) to try to find bugs in your source program that compilers often cannot detect (such as divisions by zero, null pointer dereferences, or array accesses out of bounds). These strong analyses might require a lot of time and memory to check your source program though. Static checkers simply report errors and do not generate any code.

- Interpreters: instead of producing a target program as the result of a translation, an interpreter directly performs the operations described in the source program. Interpreters are very common in computer science, from command line interpreters to PostScript interpreters embedded inside printers. Interpreters use the exact same kind of front end as compilers do. The back end is missing though, since there is no need for the interpreter to generate code ("just in time compilation" blurs the line between interpreters and compilers though). Since interpreters use the same kind of front ends as compilers do, you will first create small interpreters in your programming assignments, before moving to creating compilers.

Analysis comes in three phases:

- Lexical analysis, which reads from left to right a stream of characters making up the source code of a program and groups these characters into *lexemes*. Each lexeme belongs to a single category of lexemes which is represented by a *token*.

- Syntax analysis, where tokens are grouped hierarchically into nested collections of tokens to get a parse tree, which is a data structure inside the compiler that represents the structure of the source program being compiled.

- Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully (variables are defined before being used, for example, or variables are used with the correct types).

Lexical Analysis

The lexical analyzer (also called the *lexer* or *scanner*) takes as input a stream of characters and slices this stream of characters into a stream of basic lexical units called *lexemes* ("lexical elements"). This is a same as taking an English sentence (a string of characters) and slicing it into words. In a programming language, the lexemes include the user-defined identifiers (variables, function names, etc), the literals (constants directly written into the program), the operators (like `+`), the words reserved by the programming language (e.g. `for`, `if`, etc.), and so on. The lexer also classifies the lexemes into *tokens*, which are categories of lexemes.

For example, the statement:

```
index := 2 * count + 17;
```

is made of 24 characters (including spaces) that a lexer will slice into 8 different lexemes. For each lexeme there is a token indicating the category to which the lexeme belongs. Looking at the 8 tokens corresponding to the 8 lexemes, we see that the 8 lexemes belong to 6 different types of tokens ("identifier" and "integer_literal" each appears twice):

Lexeme	Token
`index`	identifier
`:=`	equal_sign
`2`	integer_literal
`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	integer_literal
`;`	semicolon

Note how lexemes are simply pieces of the original program, while tokens identify the category of things to which each lexeme belongs.

In general lexers use white spaces and line breaks as lexeme separators but otherwise ignore them (Python being a well known exception, since it uses source code indentation to determine the block structure of programs) and comments are simply discarded.

Tokens (categories of lexemes) are specified using *regular expressions*. Indeed, in the programming assignments, you will use regular expressions to specify lexers which will then be automatically generated by tools.

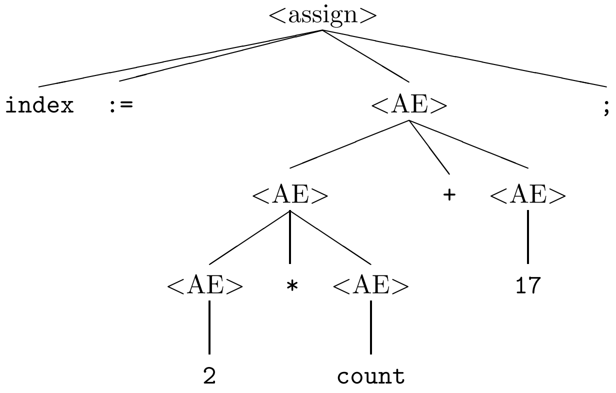
Syntax Analysis

From the stream of tokens computed above, a *parser* uses a *grammar* to analyze the stream of tokens and create a *parse tree* that represents the *grammatical structure* (the form) of the stream of tokens. This is the same as taking an English sentence and using an English grammar to check that the nouns, verbs, adjectives, etc., all appear in the correct order, and then constructing a tree that represents the structure (the form) of the sentence.

For example, using the following grammar:

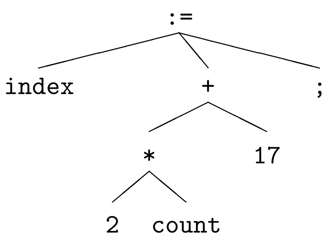
$$\begin{array}{lcl}
\langle\text{assign}\rangle & \rightarrow & \text{identifier}\ \texttt{:=}\ \langle\text{AE}\rangle\ \texttt{;} \\
\langle\text{AE}\rangle & \rightarrow & \langle\text{AE}\rangle\ \texttt{+}\ \langle\text{AE}\rangle \\
 & | & \langle\text{AE}\rangle\ \texttt{*}\ \langle\text{AE}\rangle \\
 & | & \text{identifier} \\
 & | & \text{integer\_literal}
\end{array}$$

the parser would read the expression "`index := 2 * count + 17;`" above and generate the following parse tree:



(to be more precise, we will see later that the parser would get the sequence of tokens "identifier equal_sign integer_literal mult_op identifier plus_op integer_literal semicolon" from the lexer and then would use these tokens in the parse tree instead of using lexemes like "index" or "17", but for clarity here we use the lexemes instead of the corresponding tokens).

Later we will often compress such a parse tree into the following smaller representation, called a *syntax tree*, where all the nonterminals have been removed:



(again, we will see later that a real syntax tree uses tokens, not lexemes).

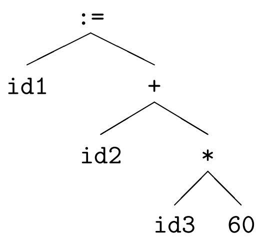
The division between lexical and syntactic analysis is somewhat arbitrary. If a construct in the programming language is atomic and can be recognized by a simple linear scan (a simple string comparison) then the lexical analysis is the obvious time to convert that construct into a token. If a construct has a more complex structure, and especially if it has a recursive structure, then it is processed during the syntax analysis. Regular expressions cannot handle recursion so recursive constructs can *never* be processed by the lexical analysis. The grammars used by parsers during the syntax analysis are often *context-free grammars*, which are the simplest form of grammars that allow for recursion.

Semantic Analysis

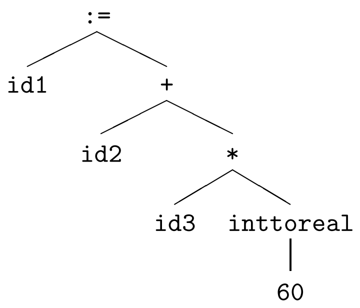
The syntax of a programming language is the *form* of its expressions, statements, and program units. The semantics of a programming language is the *meaning* of those expressions, statements, and program units. The job of the semantic analysis is then to take a parse tree as input (coming from the output of the parser) and to check that the meaning of this parse tree is correct. This is the same as taking an English sentence and checking its meaning to make sure the sentence actually makes sense.

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phases (the back end). Type checking is an important part of checking for semantic errors (only if the programming language has a type system, of course): checking that every operator and every function in the program receives values of the right types and produces values of the right types. Other common semantic checks include checking for undefined variables, checking for variables defined multiple times, and checking for uninitialized variables (variables which are used before they are given a value).

Not only does the semantic analysis check for errors (for example trying to assign a string to an integer variable) but in many languages it also has to discover where in the code *type conversions* have to be done. For example, given the following input syntax tree (based on the Dragon Book, page 7):



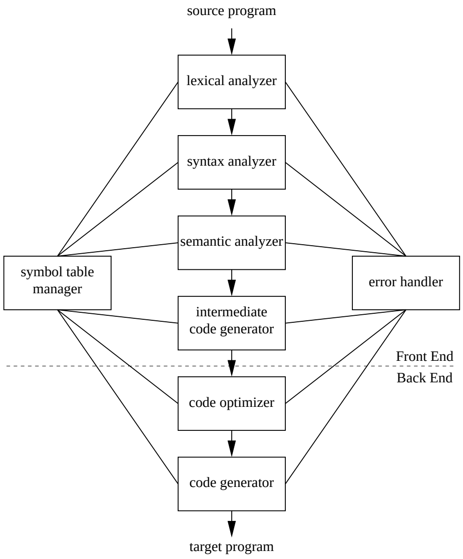
and assuming that the variables corresponding to `id1`, `id2`, and `id3` are declared elsewhere as floating point variables, then the semantic phase has to insert a type conversion in the parse tree to convert the integer `60` into the floating point number `60.0`:



The reason such type conversions have to be inserted by the compiler is because microprocessors do not have hardware to multiply (or add, or subtract, or divide) integers and floating point numbers together, they only have hardware to multiply integers with integers or floating point numbers with floating point numbers. So when the user writes in his program a multiplication between an integer and a floating point number the compiler then knows it has to introduce a type conversation in the program to transform the integer into a floating point number. Without the type conversion the code generation phase of the compiler would not be able to find a microprocessor instruction to implement the multiplication between integer and floating point number, and it would therefore have to reject the user's program. The user would then have to rewrite his program to use `60.0` instead of `60`, or to put explicitly a call to `inttoreal` in his code. This would not be difficult to do but it would be a minor inconvenience for the user, so programming languages like C or Java try to be nice with the user and require that the compiler automatically introduces a type conversion where required (the type conversion is then said to be *implicit*). Note that the compiler for a programming language like OCaml does reject programs that mix integers and floating point numbers in the same expressions, and OCaml programmers do not seem to mind much, so programming languages do not always have to be nice. In fact the OCaml programming language even has different operators for integers (`+`, `-`, `*`, etc.) and floating point numbers (`+.`, `-.`, `*.`, etc.) and programmers have no problem deciding which ones to use in a given expression, so, while introducing implicit type conversions in a program is not very difficult to do for a compiler, it is required by some programming languages only because the designers of those programming languages wanted to give the users a bit more freedom in mixing data types without having to write type conversions explicitly in their programs.

Compiler Phases

Here is a more precise description of the structure of a compiler (from the Dragon Book, page 10):



The first three phases represent most of the front end and are described above. The intermediate code generator is a small phase that is part of the front end and is described below. The remaining two phases form the back end of the compiler. Two parts of the compiler, the symbol table manager and the error handler, interact with all the other parts of the compiler. There might also be an extra phase that does machine-dependent code optimization after the target code has been generated, though we will probably not talk much about that in this course, since such optimizations are very dependent on the architecture of the microprocessors considered.

Symbol Table Manager

The *symbol table* is a data structure inside the compiler that records the various *attributes* of all the identifiers present in the source program. When an identifier (variable name, function name, class name, etc.) is detected by the lexical analysis, the identifier is entered in the symbol table. The subsequent analysis phases will then enter in the symbol table different attributes for each identifier. These attributes typically include the amount of memory allocated for an identifier, its type, its scope (where in the program it is valid to use the identifier), and the line number where it appears in the source program (for debugging purposes). In the case of functions, extra information is also added to indicate the number and type of arguments and the type of the value returned, if any.

Most of that information is computed and used by the semantic analyzer plus the various phases in the back end of the compiler. For example the semantic analyzer will compute the type of variables in a C-like programming language by looking at the definitions of the variables. It will store that information in the symbol table where it will be easily accessible later when the semantic analyzer needs to know the type of the different variables used in an expression in order to check whether the expression is correct or not.

The phases in the back end of the compiler will for example use the symbol table to store information about the amount of memory that should be allocated on the stack for local dynamic variables. That information will be necessary when the code generator generates the code that implements a procedure call.

Error Handler

Each phase in a compiler can encounter errors. After detecting an error, a phase must somehow deal with the error so that compilation can continue and more errors can be detected. A very simple compiler could stop at the first error encountered, but a compiler is generally more useful if it can detect as many errors as possible at once.

Most of the errors detected by a compiler are detected by the front end since the front end does most of the analysis of the source code. The various phases in the front end detect different kinds of errors.

- The lexical analyzer will detect errors when the stream of characters from the source program does not form any token. For example an integer cannot contain a letter so, if a program contains a sequence of characters like `12a34`, the lexical analyzer will not be able to find any token that can represent such sequence of characters and a lexical error will therefore have to be reported to the user.

- The syntax analyzer will detect errors about stream of tokens that do not follow the normal structure of programs. For example assignment statements are required to have an identifier on the left side of the assignment operator, which prevents assigning a value to an integer, so the following sequence of tokens will not be considered valid by the parser: “integer_literal equal_sign integer_literal semicolon” (e.g. “`3 := 5;`”) even though each token in isolation is correct (i.e. the lexer does not detect any error at the token level). The parser will also detect errors about missing semicolons or unmatched parentheses.

- The semantic analyzer will detect errors about well-formed programs that do not have any valid meaning. For example trying to multiply two strings in C, or trying to use the name of an array as if it were a procedure.

Note that in practice the error handling code is in fact spread among the various phases of the front end, and each phase uses a specific strategy to try to recover from errors detected by the analysis in that phase, but leads to better error recovery.

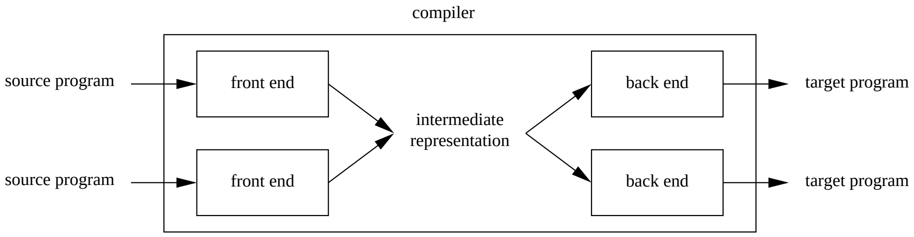
Intermediate Code Representation

After syntax and semantic analysis, many compilers generate an explicit intermediate representation of the source program that resembles low-level assembly code. There are two main reasons to do this.

First, since our ultimate goal is to generate assembly code, and since parse trees are very abstract compared to assembly code, we use the intermediate code as a representation that is closer in shape to assembly code but for which we do not have to commit to too many machine-specific details yet (like the number or registers, or the latency of specific operations). This in essence transforms the abstract syntax represented by the parse tree into abstract code for an abstract microprocessor, which gets us half the way to assembly code.

Second, we use the intermediate representation as a way to separate the front end and the back end of the compiler. Since the intermediate language is independent of the source language, a compiler can have several front ends for different source languages (e.g. GCC has front ends for C, Ada, Fortran, etc.) that all generate code in the same intermediate language. This is a simple and very efficient way for a compiler to support many source languages. This also means that several back ends for different target languages (e.g. different microprocessors) can accept the same intermediate language as input, which means the compiler can compile code for many different machines (e.g. GCC supports at least half a dozen different machine architectures). This allows compiler developers to multiply the number of front end and back end combinations without increasing the complexity of the compiler too much.

For example, by writing six front ends and six back ends, a compiler writer can create a compiler that supports thirty six possible combinations of programming languages and microprocessors. Supporting all those combinations with separate compilers would mean creating thirty six different compilers, while by the judicious use of an intermediate language the compiler writer has only to worry about twelve compiler components. The following figure illustrates the idea with two front ends and two back ends:



A good intermediate language should have the following qualities:

- It must be convenient to produce after the semantic analysis.
- It must be convenient to translate into real assembly language for all the desired target machines.
- Each construct must have a simple and clear meaning, so that optimization phases can be easily written to rewrite the intermediate code.

One common form of intermediate code is *three-address code*, which is like the assembly language of a virtual machine where each memory location can act like a register (or, equivalently, where there is an infinite number of registers that can be used like regular memory). Three-address code is a sequence of operations, where each operation has at most three operands: two sources operands and one destination operand (on the left side of the assignment operator).

For example an expression like `id1 := id2 + id3 * 60`, where all the `idX` variables are declared as floating point variables, can be translated like this (or, more precisely, the corresponding syntax tree can be translated like this, since the semantic analysis produces a parse tree as output):

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

In such code, each line has at most one operator (in addition to the assignment operator) and each operator has at most three operands. For example the `inttoreal` operator, which converts an integer into a floating point number, has two operands: one integer source operand (`60`) and then one destination operand, which is represented by the assignment to `temp1`). The `+` operator has three operands: two source operands (`id2` and `temp2` here) and one destination operand (`temp3`).

This means the intermediate code generator that generates this code has to do two things:

- decide on an order in which operations have to occur (multiplication before addition, in the example above);
- generate names for intermediate results (all the `tempX` names in the code above.

Note that the intermediate code representation is often used only in compilers that compile source code into low-level machine code. Compilers that compile one high-level programming language into another high-level programming language often just use syntax trees for intermediate representation: it is usually a bad idea to use a low-level intermediate code representation when going from one high-level programming language to another one, since going to the lower level means losing a lot of information about the overall structure of the code (intermediate code is “flat”, being just a sequence of operations written in the low-level intermediate code, so it has less internal structure than a syntax tree).

Code Optimization

The code optimization phase tries to rewrite the intermediate code to provide “better” code, with “better” often meaning “faster” (though some optimization try for example to minimize the size of the code instead of its speed—this is especially true for code that has to run on embedded microprocessors like cell phones—while some other optimizations try to minimize the number of transistor state transitions in a microprocessor to reduce the heat generated).

For example the intermediate code above can easily be rewritten like this:

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

where the integer constant has been replaced with a floating point constant (thereby necessitating no conversion) and useless intermediate variables have been removed.

Code optimization is a very complex topic, and there is no limit to the amount of time a compiler can spend trying to optimize code: some compilers are known to take hours, if not days, when compiling code with full optimizations turned on. In general though there are a few simple optimizations that usually improve code a lot without requiring too much implementation effort and without taking too much time to compute.

Code Generation

Once the intermediate code has been optimized, real machine code has to be generated (or, in the case of a high-level to high-level programming language compiler, code in the high-level target language has to be generated).

To generate low-level microprocessor code (or more often to generate assembly code, which is then translated into binary machine code by a simpler compiler called an assembler) the code generator has to make several decisions. First it has to decide where in memory each variable will go (static memory for global variables and program constants, stack memory for local variables). Then low-level instructions have to be selected to implement the various operations done in the intermediate code, and an order for the execution of these low-level instructions has to be determined. The code generator also has to select which values will reside in which microprocessor register and when. *Instruction selection*, *instruction scheduling*, and *register allocation* are very complex processes and today it is in these three areas (and in compiler optimization) that most of the research occurs. It is fairly easy to generate correct code, but it is *much* more difficult to generate correct code that is fast.

In fact instruction selection, instruction scheduling, and register allocation are all *NP-complete*. This roughly means that, for a given piece of intermediate code, there is no simple way to compute the best sequence of microprocessor instructions with the best allocation of registers for that piece of code (to be more precise: there is no polynomial-time algorithm that will directly give you the best possible machine code when given the intermediate code as input). The only way to generate the best possible machine code is to try all the possible variations and run them one by one to find which one works the best. Since the number of possible machine codes is *exponential* in the size (number of instructions) of the intermediate code, trying all the possible machines codes takes an exponential amount of time, so it is infeasible in practice.

Instruction Selection

The reason instruction selection is very complicated is because there are many ways to compute a given result. For example multiplying an integer by two can be implemented as either a multiplication by two, or as an addition of the integer with itself, or as a register left-shift (since integers are represented as binary numbers). Multiplying the same integer by three can only be done either as a multiplication or as two additions though, not as a left-shift. Multiplying the same integer by the some unknown number can only be done as a multiplication. If at runtime the unknown number turns out to always be two, then using a multiplication might be sub-optimal.

In general there are many different ways to compute the same result using different machine codes (an exponential number of different ways, in fact) and determining which way is the best is very difficult because it depends on the exact kind of computation that is being done, on the exact amounts of time required by a microprocessor to execute the different machine codes (e.g. whether a left-shift of an integer register takes more time or less time than an integer addition, etc.), as well as on which exact values are involved in the computation. The situation then becomes even more complex when one takes into account if-then-else tests, loops, function calls, etc. This results in an enormous number of possible low-level microprocessor instruction combinations to compute a given value, with no simple way to find which one is the best.

Instruction Scheduling

Even for a given way to compute an expression, there are usually many different orders in which the same low-level microprocessor instructions can be scheduled (put together in a sequence) to compute the same result, resulting in different performances. For example, imagine a microprocessor that has some integer multiplication hardware that takes two clock cycles to compute a multiplication, using a two-stage pipeline (for example, one stage to compute the various bit multiplications proper, and one stage to propagate the carries). On such a microprocessor, the following code:

```
MULI R1, R2
MULI R3, R4
MULI R2, R5
```

can be executed in a total of four clock cycles. At time zero the first multiplication `MULI R1, R2` is started. After one clock cycle that multiplication moves to the second stage of the integer multiplication hardware and the second multiplication can be immediately started using the now-free first stage of the integer multiplication hardware. After two clock cycles the second multiplication moves to the second stage of the multiplication hardware while at the same time the result of the first multiplication becomes available. That result can then directly be used as an operand in the third multiplication, which can be immediately started using the now-free first stage of the multiplication hardware. After three clock cycles the result of the second multiplication is available, and the third multiplication moves to the second stage of the hardware. After four clock cycles the result of the third multiplication is available.

Clock cycle	1	2	3	4
Stage 1	`MULI R1, R2`	`MULI R3, R4`	`MULI R2, R5`	
Stage 2		`MULI R1, R2`	`MULI R3, R4`	`MULI R2, R5`

Now, since the second multiplication is independent of the first and third multiplication (i.e. its operands and results are unrelated to the operands and results of the other multiplication) one would think that the following code is equivalent to the code above:

```
MULI R1, R2
MULI R2, R5
MULI R3, R4
```

where the order of the second and third multiplication have been swapped. In fact this code is not entirely equivalent to the code above. The reason is that now the timing is completely different. At time zero the first multiplication is started, as before. After one clock cycle that multiplication moves to the second stage of the integer multiplication hardware, as before. But now the microprocessor cannot start executing the second multiplication using the now-free first state of the multiplication pipeline. That's because the operand `R2` of that multiplication is supposed to receive the result of the first multiplication, and the first multiplication is not done yet. Microprocessors that use pipelines have special hardware to detect such conditions: in order to preserve the correctness of the whole computation the microprocessor has no other choice than to stall (wait idle) for one full cycle so that the first multiplication has time to finish. After two cycles the result of the first multiplication then becomes available and the microprocessor can then start executing the second multiplication, which was delayed in the previous cycle. The second and third multiplications will then complete as before. The total time to execute this code is therefore five cycles, not four as above, because of the extra idle cycle that the microprocessor has to introduce in the execution of the program to preserve correctness, which itself is due to the dependence between the first and second multiplications in the code above.

Clock cycle	1	2	3	4	5
Stage 1	`MULI R1, R2`	stall	`MULI R2, R5`	`MULI R3, R4`	
Stage 2		`MULI R1, R2`	stall	`MULI R2, R5`	`MULI R3, R4`

Note that some modern microprocessors are actually capable of automatically re-ordering microprocessor instructions just as they are about to be executed (e.g. they are capable of automatically transforming the second sequence of multiplications above into the first one, while the program is running) but in general it is better for the compiler to make such decisions, since the compiler can spend much more time than the microprocessor to decide which order of microprocessor instructions is the best one (the re-ordering capabilities of even the most recent microprocessors are limited to just a few instructions ahead of time, because the decision to re-order code has to be made at least as fast as the code is executed).

Such ordering problems are also made more complicated by the fact that in recent microprocessors most of the hardware functional units (like the integer unit and the floating point unit) are pipelined, so ordering decisions have to made by the compiler for multiple pipelines at the same time. In quite a few modern microprocessors even `goto`s are pipelined: the actual jump from one place in the code to another occurs only one or two cycles after a `goto` instruction starts executing, which means the microprocessor will continue executing the one or two instructions that are directly after the `goto` instruction in the old part of the code before it starts executing instructions from the new part of the code! For example the following pseudo-code (where `L` is a label that can be used as the target of a `goto` instruction):

```
  i = 3
  goto L
  i = i + 1
  ...
L:
  z = 7
  j = i
```

might lead to the following behavior in a microprocessor with a pipelined control (branching) unit with two stages:

Clock cycle	1	2	3	4	5	6
Stage 1	`i = 3`	`goto L`	`i = i + 1`	`z = 7`	`j = i`	
Stage 2		`i = 3`	`goto L`	`i = i + 1`	`z = 7`	`j = i`

Even though the instruction "`i = i + 1`" appears after the `goto` in the program, that instruction is still executed because the `goto` really only takes place in the second stage of the pipeline, so after the instruction "`i = i + 1`" has already started executing. This in turn results in the final value of `j` being 4, not 3 as expected... Again, such complications rapidly result in an instruction scheduling problem which is extremely complex and for which no nice algorithmic solution exists.

Register Allocation

The reason register allocation is very complicated is because there is a tradeoff between keeping values in registers to easily re-use the values in the future, versus re-using the register to have more space for other computations (since the actual operations in a modern RISC microprocessor occur from register to register only). For example, if a value has just been computed and used by the program, and that same value has to be used some time later again, is it more efficient to:

- keep the value in a register until it is needed again, thereby reducing the number of registers available for other computations that have to be done in the meantime;
- move the value into some memory location, which frees the register but takes some additional time since it requires one extra memory write to save the value into memory and one extra memory read later to get the value back (and memory is slow compared to the microprocessor);
- simply forget the value and recompute it from scratch later when we need it again, which again frees the register but requires extra computation later?

For example, given the code (using intermediate code here):

```
y = 2 * x
z = 2 * y
...
w = 3 * y
```

is it better to keep the value of `y` in a register once it has been computed until it is used again in the computation for `w`, or is it better to store `y` in memory after the computation of `z` and load it back from memory into a register before it is used for computing `w`, or is it better to simply discard the value of `y` after `z` has been computed and to recompute `y` from scratch just before computing `w` (assuming the value of `x` itself is still available in one way or another)? The answer is that it depends on how many instructions are represented by "`...`": if there are no instructions between the computation of `z` and the computation of `w` then it is probably better to keep `y` in a register, while if the two computations of `z` and `w` are separated by hundreds of instructions then it is probably better to free the register used by `y` for some other intermediate computation. The decision also depends on the respective costs of keeping `y` in a register, storing and loading it from memory, or recomputing it from scratch. Such choices have to be made for every value stored in a register, which means again that an enormous number of combinations is possible and there is no simple way to find the best one.

Note that both instruction selection and instruction scheduling influence register allocation: different sequences of instructions, or different orders for the same instructions, might result in fewer registers being required to carry out some computation. So the three problems of instruction selection, instruction scheduling, and register allocation are not independent.

In Practice...

Compilers have therefore to resort to using *approximate* instruction selection, instruction scheduling, and register allocation algorithms that generate code which is not always the theoretically best code but which is often good enough in practice.

From the optimized intermediate code of the previous section the code generation phase will for example generate the following assembly code:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

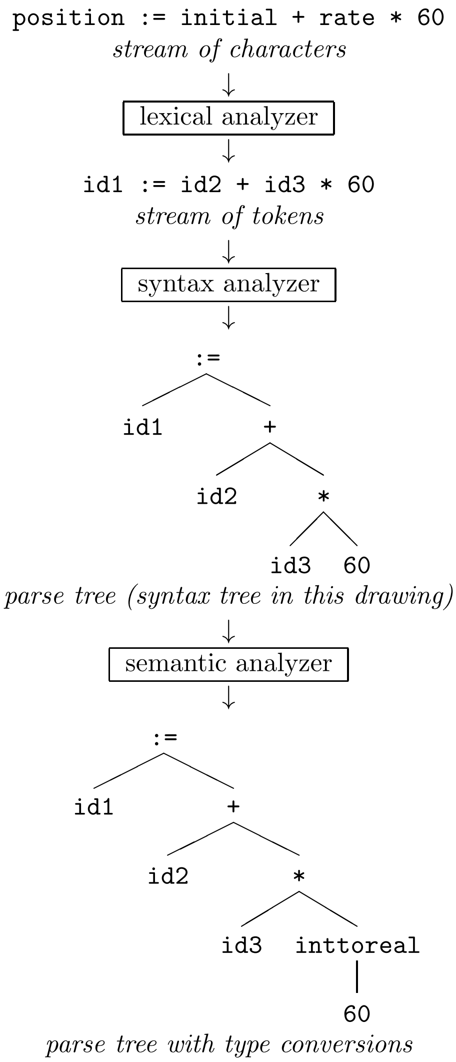
Note that the low-level microprocessor instructions selected depend on the type of the variables (`ADDF` is a floating point addition, which is different from the integer addition `ADDI` at the level of the microprocessor, even though the intermediate code above uses the `+` symbol for both). Microprocessor instructions also often do not have an explicit destination, because the size of an instruction, once encoded into binary format (32 bits on a 32 bits machine, for example) is too small to encode the names of more than two registers. For example the `ADDF` instruction above adds the content of registers `R2` and `R1` and implicitly puts the result back into register `R1`, so the code generator has to keep track of that. Finally all operations have to be done from register to register, so in addition to doing the computations proper, the generated code also has to include instruction to move data from memory into registers and back, using `MOVF` operations.

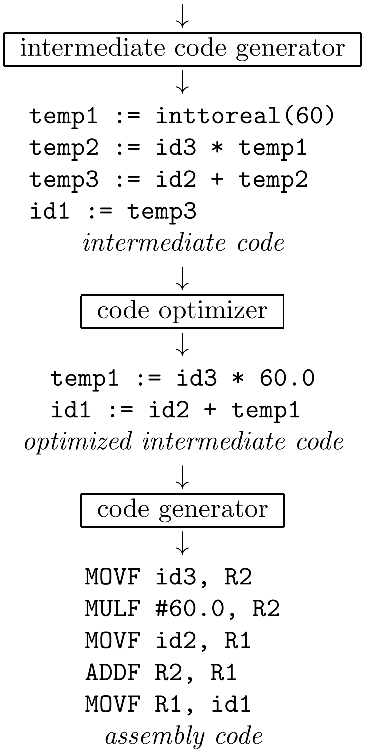
Restricting operations to being only from register to register is a characteristic of modern RISC microprocessor architectures; old CISC architectures allowed operations from memory to memory as well, without having first to move data into registers. This made it easier for compilers to generate code for CISC microprocessors, but it also required the hardware for these computers to be more complex. RISC microprocessors are (conceptually) simpler, which allows them to run faster, but generating good code for these microprocessors then requires more work on the part of the compilers. This is a general trend in hardware and compiler design: more and more of the complexity is moved from the hardware to the compiler, which makes the hardware simpler and therefore faster, but requires more complex compilers to generate good code.

Internal Code Representation

As we have seen above, different phases of the compiler use different internal representations. In general a compiler should use as few internal representations as possible, as it minimizes the amount of code that has to be implemented to manipulate the different representations. In particular it is often a very good idea to have all optimization phases accept the same kind of intermediate code and generate the same kind of intermediate code, so that multiple independent optimization phases can be easily chained one after the other.

Here is a summary of the different phases of a compiler, with examples of the kind of data structures that they use or generate (from the Dragon Book, page 13):





```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

intermediate code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

optimized intermediate code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

assembly code

In addition to all these different data representations, the symbol table also keep track of attributes (like types, or memory sizes) for the identifiers `position`, `initial`, and `rate`, as explained above.

Cousins of the Compiler

A compiler usually compiles one file at a time and generates a single file containing assembly code. The compiler therefore has to be surrounded by a series of other tools that implement the other parts of the whole compilation process.

The *preprocessor* is in charge of processing the macros in the source program (i.e. transform the syntax of the source program according to syntax rules described by macros) and taking care of file inclusion. For example, the C preprocessor will replace all preprocessor directives of the form `#include<file.h>` with the content of the header file `file.h`, for all such directives that appear in a C program.

The compiler will then compile the resulting preprocessed program, which is not anymore the original source program, but which is still a program in the original source language. The result of the compilation is assembly code.

The assembly code generated by the compiler will then be compiled by the *assembler*, which is a simple compiler that reads assembly code and converts each assembly operation (like `MOVF` or `MULF` above) into a directly equivalent binary instruction that can be executed by the microprocessor. These binary instructions (*object code*) are stored in a file called an *object file*. That object code is *relocatable*, meaning that it assumes that memory addresses start at zero and that this is where it will be placed in memory when executed, without taking into account the memory addresses of the rest of the code in the software:

0	object code

An object file containing relocatable object code is created like this for each original C source file.

After that, a *linker* takes the relocatable object code in each object file and modifies the addresses in that code so that the code can be placed at a specific address in memory (an address other than the address zero, which is what the assembler assumed). The linker does that for all the object files which compose the software to be created. The linker takes each object file in turn and places its content (object code) at a specific address in memory, while at the same time changing the addresses in the code to reflect the final location of that object code in memory. The result is that the multiple pieces of object code coming from the different object files now together form a single continuous piece of object code in memory. As it does this, the linker also copies into the program all the object code for all the basic functions provided by the programming language itself (for example the object code for the `printf` function in C) that are used by the program. The object code for all these basic functions comes from an archive of object code called a *library*. Once this is done, the resulting code is stored in a single file on the computer's hard disk, which is the complete executable program.

For example, consider two pieces of relocatable code, with the first one being 3000 bytes in size and containing a pointer `p` to address 2240, and the second one being 2000 bytes and containing a pointer `q` to address 1320:

0	object code 1 `int *p;` 2240	} 3000 bytes	0	object code 2 `int *q;` 1320	} 2000 bytes

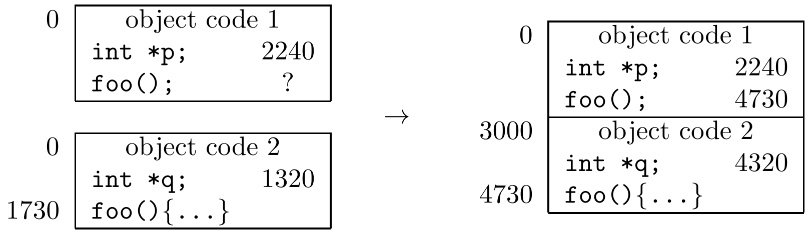
When the linker puts these two pieces of relocatable code together in a single file, the second piece of code will not start at address 0 anymore but at address 3000:

0	object code 1 `int *p;` 2240
3000	object code 2 `int *q;` 1320

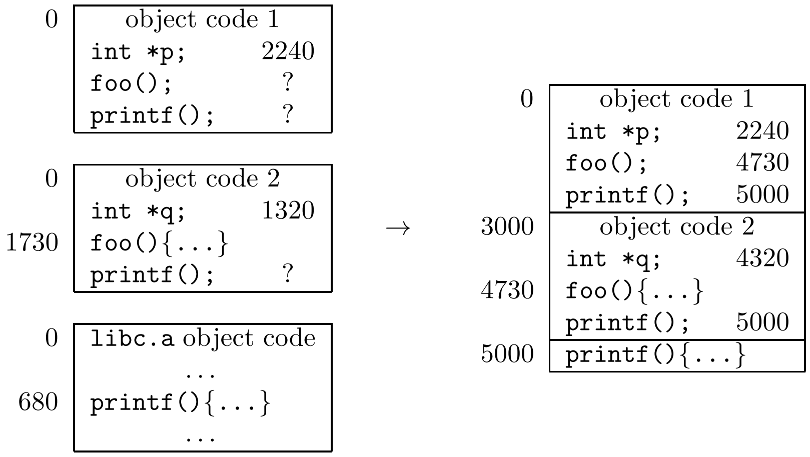
As it does this, the linker needs to change the pointer `q` in the second piece of code from address 1320 to address 4320 (1320 plus the size 3000 of the first piece of code):

0	object code 1 `int *p;` 2240
3000	object code 2 `int *q;` 4320

The linker also resolves all the references in the different pieces of object code to make sure the software contains all the necessary code and every part of the code knows where to find the other parts it needs. For example, one piece of object code that came from one object file might need to call a function `foo` which is implemented by another piece of object code that came from another object file. Once both files have been placed in memory by the linker, the linker will modify the code that came from the first file so that it knows where in memory it can find the code of the required function `foo` that came from the second file:



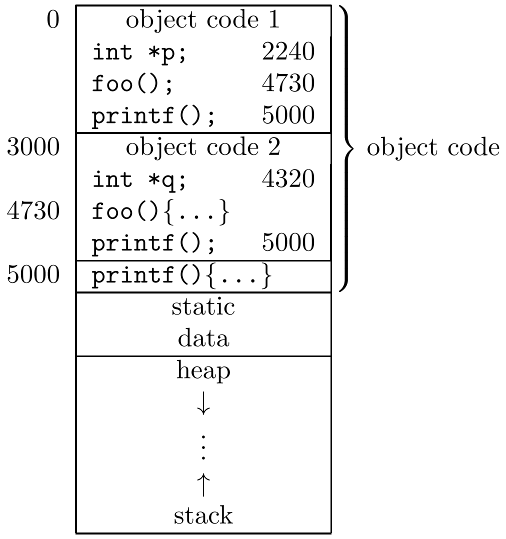
Similarly, the object code might need to call functions provided by the implementation of the programming language. For example, in C, the object code created from the user's source program needs to know where the `printf` function is in order to produce output. The linker will modify the object code generated from the user's program so that that code knows where to find the implementation of the `printf` function. The code implementing the `printf` function (which is some object code stored in an archive of object code called a library) is in fact loaded and linked with the software just like any other piece of object code.



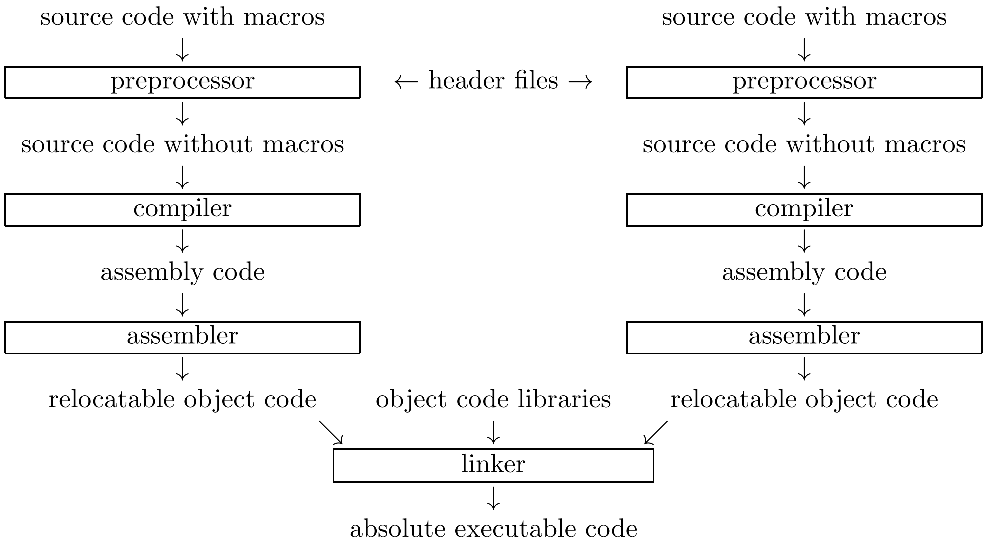
Once all the references in the object code have been resolved, the linker copies all the content of memory (the whole object code, after all the addresses have been modified appropriately) into a single file (the executable program) on the hard disk that contains *absolute* code.

Note that the resulting executable file does not contain a complete copy of the library, but contains only the parts of the library which are actually required by the user's code. In our example above, only the code of the `printf` function is copied into the executable program, nothing else, so the address for the code of the `printf` function is 5000, not 5680, because only `printf` is copied, not the code which is before or after the `printf` function in the library `libc.a`.

Later on, when the user wants to run the software, the user tells the operating system to start the program (by clicking on an icon, or typing the name of the software at a shell prompt). The operating system then creates a new process in memory and uses the *loader* to simply copy the whole content of the executable file into memory. This becomes the code part of the new process created by the operating system. The loader also initializes the static data of the process (for global variables and program constants), and creates the heap (for dynamic memory allocation, using for example `malloc`) and the stack (used for function calls; in particular the loader creates the first stack frame on the stack for the program's `main` function). The loader then transfers control of the microprocessor to the beginning of the `main` function in the code part of the program. From there the microprocessor starts executing all the instructions in the executable code in sequence and the process is running:



The following figure illustrates the whole process, with two source files being compiled and linked with libraries to get executable code.



Here is an example illustrating the different steps, using the GCC compilation system on a Unix machine. We start with the following C source program:

```
$ cat test.c
#include <stdio.h>

int main(void) {
    printf("hello\n");
    return 0;
}
```

We can use the `file` program to check what kind of content is in the file:

```
$ file test.c
test.c: ASCII C program text
```

First we run only the preprocessor on the file using the `cpp` command (or we can use the `gcc` command with the `-E` option). This gives us a big file `test.i` that still contains C code but with no `#include` directives:

```
$ cpp test.c > test.i                   # or: gcc -o test.i -E test.c
$ ls -l
-rw-------  1 meunier  users     72 Jan 25 09:58 test.c
-rw-------  1 meunier  users  11809 Jan 25 09:58 test.i
$ file test.i
test.i: ASCII C program text
```

We can now compile this preprocessed C code into assembly code using the compiler proper `cc1` (or using `gcc` with the `-S` option):

```
$ /usr/lib/gcc-lib/i386-unknown-openbsd6.4/4.2.1/cc1 -quiet test.i
```

```
# or: gcc -S test.i
$ ls -l
-rw-------  1 meunier  users     72 Jan 25 09:58 test.c
-rw-------  1 meunier  users  11809 Jan 25 09:58 test.i
-rw-------  1 meunier  users    703 Jan 25 10:00 test.s
$ file test.s
test.s: ASCII assembler program text
```

The content of the file `test.s` is assembly code for the Intel family of microprocessors, since the Unix system I used for these examples is running on a PC with an Intel Pentium microprocessor:

```
$ cat test.s
.file "test.i"
.section .rodata
.LC0:
.string "hello"
.text
.globl main
.type main, @function
main:
leal 4(%esp), %ecx
andl $-16, %esp
pushl -4(%ecx)
pushl %ebp
movl %esp, %ebp
pushl %ebx
pushl %ecx
subl $32, %esp
call __i686.get_pc_thunk.bx
addl $_GLOBAL_OFFSET_TABLE_, %ebx
leal .LC0@GOTOFF(%ebx), %eax
movl %eax, (%esp)
call puts@PLT
movl $0, %eax
addl $32, %esp
popl %ecx
popl %ebx
popl %ebp
leal -4(%ecx), %esp
ret
.size main, .-main
.section .text.__i686.get_pc_thunk.bx,"axG",@progbits,__i686.get_pc_thunk.bx,comdat
.globl __i686.get_pc_thunk.bx
.hidden __i686.get_pc_thunk.bx
.type __i686.get_pc_thunk.bx, @function
__i686.get_pc_thunk.bx:
```

```
movl (%esp), %ebx
ret
```

At the beginning of this assembly program we can see the name `test.i` of the preprocessed C program that was given as input to the compiler. Then comes the section for the read-only data ("`rodata`") of the program. This section contains the static data from the original C program, which in our example is the string constant `"hello"`. Then comes the text section (i.e. code) of the program with the assembly code for the `main` function. In the middle of that code you can see that the program manipulates the address `LC0` of the string `"hello"` just before making a call to the `puts` function to print the string to the screen: the compiler has replaced the `printf` function call in the original C code with a call to the more efficient `puts` function in the assembly code, as an optimization! The various names like `%ebp`, `%esp`, etc., are symbolic names for various hardware registers inside the microprocessor.
We now use the `as` command (or the `-c` option of `gcc`) to run the assembler on the assembly code to get a file `test.o` that contains relocatable object code for the 80386 family of 32 bits Intel microprocessors:

```
$ as -o test.o test.s                   # or: gcc -c test.s
$ ls -l
-rw-------  1 meunier  users     72 Jan 25 09:58 test.c
-rw-------  1 meunier  users  11809 Jan 25 09:58 test.i
-rw-------  1 meunier  users    952 Jan 25 10:03 test.o
-rw-------  1 meunier  users    703 Jan 25 10:00 test.s
$ file test.o
test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
```

ELF ("Executable and Linking Format") is one of the standard file formats for object code. Which format is used (among ELF, COOF, and many others) is dependent on the operating system. As indicated, the object code is relocatable object code.

The object code in `test.o` actually contains extra debugging information that lists the names of the functions and global variables defined or used in the program. We can use the Unix `nm` command on the object file to get this list of names:

```
$ nm test.o
         U _GLOBAL_OFFSET_TABLE_
00000000 ? __i686.get_pc_thunk.bx
00000000 T main
         U puts
00000000 F test.i
```

Here the letter "`T`" (for "text") means that the object file contains the code for the `main` function. The letter "`U`" (for "undefined") means that the object file uses the `puts` function but does not contain the code for that function.

We then take the object code from test.o and link it with the C standard library libc.a that contains the code for the puts function. We also link the object code with the three files rcrt0.o, crtbegin.o, and crtend.o. These three files contain special object code which is used by the operating system to start and end the program. Note that this linking step is very dependent on which compiler and operating system are used, so a different system will probably used different file names. The result of the linking is a file named test which is the program that contains the final executable code:

```
$ ld -Bstatic -o test test.o /usr/lib/rcrt0.o /usr/lib/crtbegin.o
/usr/lib/crtend.o /usr/lib/libc.a       # or: gcc -static -o test test.o
$ ls -l
-rwx------  1 meunier  users  368768 Jan 25 10:05 test
-rw-------  1 meunier  users      72 Jan 25 09:58 test.c
-rw-------  1 meunier  users   11809 Jan 25 09:58 test.i
-rw-------  1 meunier  users     952 Jan 25 10:03 test.o
-rw-------  1 meunier  users     703 Jan 25 10:00 test.s
$ file test
test: ELF 32-bit LSB shared object, Intel 80386, version 1
```

Checking the names of the functions defined in the executable program shows that the program now contains the code for the puts function (as well as the code for many other functions from the C standard library):

```
$ nm test
00000a60 t L1
000021ee t L1
[...]
000001f0 T main
[...]
00000610 T puts
[...]
00003b40 t wrterror
00000000 F wsetup.c
```

Now that we have executable code, we can finally run the program!

```
$ ./test
hello
```

Of course going through all the different steps of the compilation process one by one is painful. Fortunately compilation systems usually will run all the different steps automatically for you if you do not specify otherwise. This will allow you to go from the original C source program to the executable program in one big step without having to worry about all the various steps taken internally by the compilation system:

```
$ gcc -static -o test test.c
$ ./test
hello
```

For students interested in exploring the details of GCC's various phases, the gcc command has a -v option that makes gcc display the commands it uses internally for each step of the compilation process (that option can also be used in combination with the -E, -S, or -c options described above). For example:

```
$ gcc -v -static -o test test.c
Reading specs from /usr/lib/gcc-lib/i386-unknown-openbsd6.4/4.2.1/specs
Target: i386-unknown-openbsd6.4
Configured with: OpenBSD/i386 system compiler
Thread model: posix
gcc version 4.2.1 20070719
 /usr/lib/gcc-lib/i386-unknown-openbsd6.4/4.2.1/cc1 -quiet -v test.c -quiet
 -dumpbase test.c -auxbase test -version -o /tmp/ccerb5w9.s
ignoring duplicate directory "/usr/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/include
End of search list.
GNU C version 4.2.1 20070719  (i386-unknown-openbsd6.4)
        compiled by GNU C version 4.2.1 20070719 .
GGC heuristics: --param ggc-min-expand=99 --param ggc-min-heapsize=129834
Compiler executable checksum: 216b38549429e597949196d796483da4
 as -o /tmp/ccOZEjTJ.o /tmp/ccerb5w9.s
 /usr/lib/gcc-lib/i386-unknown-openbsd6.4/4.2.1/collect2 -e __start
   -Bstatic -dynamic-linker /usr/libexec/ld.so -L/usr/lib -o test
   /usr/lib/rcrt0.o /usr/lib/crtbegin.o
   -L/usr/lib/gcc-lib/i386-unknown-openbsd6.4/4.2.1 /tmp/ccOZEjTJ.o
   -lgcc -lc -lgcc /usr/lib/crtend.o
```

You can also use the -save-temps option of gcc if you want to see all the intermediate files generated by the different phases of the compilation process without having to run each phase one by one by hand:

```
$ ls
test.c
$ gcc -save-temps -static -o test test.c
$ ls
test  test.c  test.i  test.o  test.s
$ file *
test:   ELF 32-bit LSB shared object, Intel 80386, version 1
test.c: ASCII C program text
test.i: ASCII C program text
test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
test.s: ASCII assembler program text
```

In addition to nm, there is also another more advanced Unix tool called objdump that you can use to get more information about any relocatable object code, library, or executable program. In fact that tool can even disassemble object code back into assembly code!

To finish with this example, let's look quickly at the C standard library. In general a library is just an archive of object files, with each object file containing the relocatable object code for some functions. In the case of the C standard library, the functions defined in the library are simply all the standard functions (like `printf`, `strcpy`, etc.) that are defined by the C language. Again we can use the Unix `nm` command to look at the content of the C standard library:

```
$ nm /usr/lib/libc.a | egrep ' puts'
00000000 T puts
00000000 F puts.c
```

The letter "`T`" shows that `libc.a` contains the object code for the function `puts`. This is the code that was copied by the linker from the library into the executable `test` program when we linked `test.o` with `libc.a` above. Inside `libc.a`, the object code of the `puts` function itself originally comes from a normal C file named `puts.c` (as indicated by the "`F`" letter).
Finally we can use the Unix `file` and `ar` ("archive") commands to check that `libc.a` is indeed an archive of regular object files:

```
$ file /usr/lib/libc.a
/usr/lib/libc.a: current ar archive
$ ar tfv /usr/lib/libc.a
rw-r--r-- 0/0   4464 Jan  1 08:00 1970 putwc.o
rw-r--r-- 0/0   2823 Jan  1 08:00 1970 truncate.o
[...]
rw-r--r-- 0/0   5439 Jan  1 08:00 1970 puts.o
[...]
rw-r--r-- 0/0   1288 Jan  1 08:00 1970 munmap.o
rw-r--r-- 0/0   1259 Jan  1 08:00 1970 write.o
```

As we can see, `libc.a` is an archive file that contains, among many others, a file called `puts.o`, which is the file containing the object code for the C standard `puts` function. This file `puts.o` is itself just the result of compiling a normal C file called `puts.c`, as can be seen above in the result of the `nm` command.

In this course we will concentrate on the compiler proper, to generate assembly code from a single preprocessed source file (i.e. how to go from the file `test.i` to the file `test.s` in the example above). It is nevertheless important for you to understand the whole process from source program to executable program because in many cases you will not be able to solve compilation problems for a piece of software if you do not understand how the preprocessor, compiler, assembler, linker, and operating system loader are supposed to work together.

Static Linking versus Dynamic Linking

The C standard library is quite big:

```
$ ls -l /usr/lib/libc.a
-r--r--r--  1 root  bin  6139108 Oct 12 03:33 /usr/lib/libc.a
```

Since we saw above that the `test` executable program is 368768 bytes in size, this clearly shows that the linker does not include the whole of the C standard library into the program we created. Nevertheless 368768 bytes is still a big size for a program that only prints the string `"hello"`! There are two reasons for this.

The first reason is that the program includes a lot of information that is used only when we need to debug the program. In particular the executable program contains information from the compiler's symbol table about all the functions and global variables defined in the program (this is in fact the information that the `nm` command shows). This information is not required during normal execution of the program and we can get rid of it using the `strip` command (again, on a Unix machine):

```
$ ls -l test
-rwx------  1 meunier  users  368768 Jan 25 10:05 test
$ strip test
$ ls -l test
-rwx------  1 meunier  users  90764 Jan 25 10:18 test
```

Of course once we have done that then the `nm` command cannot list much information any more:

```
$ nm test
         W _Jv_RegisterClasses
20004e3c A __bss_start
[only about 15 lines now, with no "main" or "puts"]
```

The second reason the executable program is big is that we linked the program *statically* (as explicitly indicated by the `-Bstatic` option of `ld` in the linking step above): we ran the linker to create the executable program and the linker then copied into the program all the code necessary for its execution. That includes the code for our `main` function, the code for `puts`, the code for all the functions used by `puts`, the code for all the functions used by those functions, etc. In the end this adds up to a fair amount of code (90764 bytes, as shown above).

Now, if you think about it, linking programs like this means that *every* C program you create will contain copies of the code of many functions from the C standard library. For example, on an average Unix system, there are hundreds of programs written in C which use the `printf` function. This means that, when these programs are compiled, you end up with hundreds of executable programs that all contain a copy of the same code for `printf` (or `puts`). This makes each program big, which in turn means that all these programs take up a lot of disk space for storage as well as a lot of memory when you execute them! Nowadays disk space is cheap but memory is still in high demand in computers so statically linking programs is not the best solution.

Instead, most programs today are *dynamically* linked. This means that, when the relocatable object code from `test.o` is given to the linker, the linker puts that object code into the executable program as usual, but the linker does *not* copy into the program the relocatable object code from the libraries (the code for `puts` for example). Instead the linker just puts in the executable program some information indicating where the library code is supposed to come from. Later, when the user executes the program, the loader will copy the executable program from the hard disk into memory and, rather than immediately transfer control to the `main` function of the program, the loader will first use the linker to re-link the program. The linker in turn will use the information that was stored in the executable program to know directly which library code the program has to be linked with. Once this linking is done then control will be transfered to the `main` function of the program as before and the process will start executing as usual. So in this case the linking will only really happen right before the program starts executing inside the process, hence why such a program is called dynamically linked.

Dynamic linking has several advantages.

- All the hundreds of executable programs you have on your computer do not longer each need to contain a copy of the code for the `printf` function (for example). The code for that function only needs to be stored once on the hard disk, as part of the library file, and you therefore save quite a lot of disk space.

- Since the executable programs are smaller the loader takes less time to copy them from the hard disk into memory when the user wants to run them.

- If the linker and the operating system coordinate their actions then it is possible to have only a single copy of the library code in memory, even when multiple programs need this code to execute. The loader can load the code of the library into physical memory once and the operating system can then manipulate its virtual memory system (the page tables for the various processes) so that this library code appears in the virtual memory address space of each process that requires the library. The linker then just has to dynamically link the program code with the library code in the virtual address space and the program can then start running. This means that independent programs running inside independent processes can share the same library code in memory. This code sharing is probably the most important advantage of the dynamic library system since it saves a lot of memory space.

- When a program is loaded into the memory of a process by the loader, the linker has to find all the dynamic libraries that the program requires and load these libraries from the hard disk into memory before the program can be re-linked. This can take some time, but, since many processes can share the same library code in memory, in practice very often a library required by a process will already be present in memory simply because another process is already using it. In that case the linker does not have to re-load the library from disk and can instead directly link the library that is already in memory with the code of the program that has just been loaded by the loader. This often saves a lot of time, particularly for libraries like the C standard library that are used by many programs. It is only when an executable program needs a library which is seldom used that the linker will have to load the library from the hard disk (to speed up things even more it is in fact even possible to start running the program immediately and only load the required libraries later, on demand. . .)

Dynamic linking is not perfect though, and has several disadvantages.

- A program has to be re-linked every time it is loaded into memory before it can start executing. This wastes some time. In practice though linkers are optimized to speed up the dynamic linking (by using caches, for example) and the format of dynamic libraries is optimized as well (by using special symbol tables, for example) to make the linking as simple and as fast as possible. So in most case the dynamic linking does not take too much time (except if you need to link many different programs that use many different libraries, all at the same time) and this small amount of time wasted is well compensated for by the speed gains coming from sharing commonly-used libraries between processes.

- If a program requires a library which is not already in memory then the linker has to spend some time finding the library on the hard disk and loading it into memory. Again this is not really a noticeable problem in practice unless you try to execute a program that requires many libraries that are seldom used by other programs.

- The whole linking and loading system is much more complex. This is why, historically, program were linked statically, simply because it was easier to do. Dynamic linking and library code sharing between processes require the cooperation of the operating system and is more complex to implement than simply statically putting all the necessary library code into a single complete executable program. Fortunately users do not have to worry about the complexity of dynamic linking, only programmers implementing linkers and loaders have to worry about that. From the point of view of the user the dynamic linking system is mostly invisible.

- The libraries themselves have to be in a special format to allow for dynamic linking. So, for example on a Unix system, `libc.a` is the C standard library that is used for static linking, but if you want your program to be dynamically linked then you should use a different C standard library:

  ```
  $ ls -l /usr/lib/libc.so.92.5
  -r--r--r--  1 root  bin  3583962 Jan 25 08:34 /usr/lib/libc.so.92.5
  $ file /usr/lib/libc.so.92.5
  ```

```
/usr/lib/libc.so.92.5: ELF 32-bit LSB shared object, Intel 80386, version 1
```

The `.so` file extension means that this library has been compiled in a special way to be used dynamically by the linker and be shared by multiple processes in memory (hence the name “shared object” in the output of the `file` program and the `.so` file name extension of the library). You cannot use `libc.a` for dynamic linking and you cannot use `libc.so` for static linking. The two libraries are created starting from the same C code for the same library functions, but the internal formats of the two libraries are different.

- Shipping software to customers becomes more complex. If you use dynamic linking for some software you are writing, then, when shipping that software to a customer, you have to remember to send both the executable program as well as all the dynamic libraries required by the program. If you forget to send the dynamic libraries then your customer will get an error message like “dynamic library not found” coming from the dynamic linker on his computer when he tries to execute your program (unless of course he happens by chance to have the exact same dynamic libraries on his computer as you do on your computer). There is no such problem with executable programs that are statically linked, since these programs contain all the necessary library code.

- Similarly, if you mistakenly delete some dynamic library on your computer then all the programs that require this library will stop working. This means for example that, if you accidentally delete `libc.so` on your computer (the dynamic version of the C standard library), then all your dynamically linked C programs will stop working. On a Unix system this means that most programs will stop working and your system will become unusable (try it, it’s fun!) This is why on Unix systems all the basic system programs (usually found in the `/bin` and `/sbin` directories) that are necessary in case of a system emergency (like the programs to rebuild corrupted file systems, configure network interfaces, recreate special devices, mount disks, etc., as well as the command line interpreter of the system administrator) are normally all linked statically.

- Similarly if you delete the dynamic linker itself! On Unix systems it is usually called `ld.so` and if you delete it then you cannot run any dynamically linked program anymore, even if you have all the required libraries for the program. Note that, as its name indicates, the dynamic linker `ld.so` is not a separate program but is instead itself a dynamic library and so it has to be dynamically loaded into memory before it can be used to dynamically re-link normal executable programs. So who loads (and links) the dynamic linker in the first place? The answer is: the kernel. This is done as part of the operating system’s boot sequence.

- Since the code of the libraries is not included in the executable program, the dynamic linker has to be very careful about dynamically linking the program with the right versions of the libraries. For example, if, when creating an executable program, the static linker `ld` puts in the program some information about version 92 of the C standard library, then when the program is executed the dynamic linker `ld.so` should link the program only with version 92 of the library. If the C standard library is then upgraded from version 92 to version 93 and the old version 92 is removed, all the programs that require the old version of the library will stop working. This is not a problem with statically linked programs since they directly contain a copy of the code of the library so changing the library to a new version does not affect the library code that had already been previously copied in the executable program.

The reason the dynamic linker cannot use the new version of the library is because the new version might have removed some functions from the library, or might have changed the parameters of some of the functions in the library, etc. Minor changes in the library can be handled by dynamic linkers but any major change about how the library has to be used will result in the new version being incompatible with the old one.

The result of all this is that dynamic libraries have to have a version number (like the `92.5` in `libc.so.92.5` above) while static libraries do not need one (like `libc.a` above). It also means people have to be very careful about not deleting from their computers the old dynamic libraries that are still required by some programs, even if newer versions of the same libraries are available on the same computer. In the long term it means that a computer might end up with having many different versions of the same dynamic library and the system administrator of the computer who wants to clean the hard disk has to be careful about which libraries can be deleted and which ones cannot. The software engineer creating a program also has to make sure that he sends to his customer the right versions of all the dynamic libraries required by his software. This slowly turns into what is called "DLL hell" where someone has an executable program and many different versions of all the dynamic libraries, except for the right versions of the dynamic libraries that are required to make the program work ("DLL" means "Dynamic-Link Library" and is the name given by Microsoft to its dynamic libraries; the same problem can also exist on Unix systems of course: Linux is one example where this problem occurs fairly often because many users exchange executable programs but forget to also exchange the associated dynamic libraries).

In practice the advantages of dynamic linking far outweigh the disadvantages and therefore most systems today will automatically create dynamically linked executable programs, unless explicitly told otherwise.

On a Unix system you can use the `ldd` commands to test whether a given executable program has been linked statically:

```
$ ld -Bstatic -o test test.o /usr/lib/rcrt0.o /usr/lib/crtbegin.o
/usr/lib/crtend.o /usr/lib/libc.a        # or: gcc -static -o test test.o
```

```
$ ls -l test
-rwx------  1 meunier  users  368768 Jan 25 10:23 test
$ ldd test
test:
        Start    End      Type  Open Ref GrpRef Name
        00fdc000 20fe5000 dlib  1    0   0      test
$ ./test
hello
```

or whether the executable program has been linked dynamically:

```
$ ld -Bdynamic -dynamic-linker /usr/libexec/ld.so -o test test.o
/usr/lib/crt0.o /usr/lib/crtbegin.o /usr/lib/crtend.o /usr/lib/libc.so.92.5
$ ls -l test
-rwx------  1 meunier  users  10295 Jan 25 10:25 test
$ ldd test
test:
        Start    End      Type  Open Ref GrpRef Name
        14af8000 34afb000 exe   1    0   0      test
        0eef7000 2ef25000 rlib  0    1   0      /usr/lib/libc.so.92.5
        0277f000 0277f000 ld.so 0    1   0      /usr/libexec/ld.so
$ ./test
hello
```

As you can see the non-stripped statically-linked executable program is 368768 bytes in size while the non-stripped dynamically-linked version is only 10295 bytes long. Such difference is to be expected since the the second version does not include any library code. This second version requires both the `libc.so.92.5` dynamic C standard library and the dynamic linker `ld.so` to be present on the computer though, otherwise it will not be possible to execute the program.

As you can guess it is possible to make the dynamically-linked executable program even smaller by stripping from it any debugging information about the functions and global variables it uses, just like in the statically-linked case:

```
$ strip test
$ ls -l test
-rwx------  1 meunier  users  9248 Jan 25 10:25 test
$ ./test
hello
```

(it is possible to make the executable program even a little bit smaller by having the compiler optimize the code for space).

Finally, even in the dynamic linking case it is of course possible to do at once all the steps of the compilation process, from C source program to executable program, just like in the static linking case. In fact the advantages of dynamic linking are so great that it is the default behavior of the GCC compiler (hence why I had to use the `-static` option of the `gcc` command when I wanted to force `gcc` to link the program statically above):

```
$ gcc -o test test.c
$ ls -l test
-rwx------  1 meunier  users  10295 Jan 25 10:26 test
$ ./test
hello
```

Compiler Construction Tools

In addition to the usual software development tools used by software engineers, compiler writers also have access to more specific tools that make implementing compilers easier. Over time, people have created tools to automatically generate the code for some specific parts of compilers, particularly the parts that belong to the front end.

Note that these tools are tools to help you write the compiler itself. These tools automatically generate code that becomes part of the compiler, *not* part of the programs compiled by the compiler!

Here is a sample list of the kind of tools that are available.

Lexer generator. A lexer generator automatically generates the code of lexical analyzers, from specifications based on regular expressions. A lexer (lexical analyzer) generated in such a way implements in essence the finite automata corresponding to those regular expressions. The generated lexer can then be used as the lexical analyzer in the front end of a compiler. The most well known scanner generator for the C programming language is called *Lex* (a GNU version exists called *Flex*).

Parser generator. Similarly, a parser generator automatically generates the code of syntax analyzers, from specifications based on (usually context-free) grammars. A parser (syntax analyzer) generated in such a way implements in essence a pushdown automata corresponding to the context-free grammar. The generated parser can then be used as the syntax analyzer in the front end of a compiler. The most well known parser generator for the C programming language is call *Yacc* (a GNU version exists called *Bison*).

Syntax-directed translation engines. These generators automatically create tree-walking code that automatically traverses parse trees and generates intermediate code. The specification for the code generation is based on *attribute grammars.*

Automated code generators. These generators automatically generate assembly code from the intermediate code. The translation from one to the other is specified through the use of templates that represent sequences of machine instructions implementing the intermediate instructions. As we have seen above, generating good code is very difficult. Automated code generators therefore often do not generate very good code, but using such generators can be useful if one wants to simplify the code generation process a lot. It also helps ensuring the generated code is correct, since it is easier to check the correctness of templates than to check the correctness of complex instruction selection, instruction scheduling, and register allocation algorithms.

Dataflow engines. Many compiler optimizations can be described as dataflow problems. A dataflow engine computes how values in a program flow between different parts of a program. For example, given the following code:

```
int x = 5;
int y = x;
```

a dataflow analysis will compute that the value `5` flows from the term `5` into the definition of `x`, then from there into the use of `x`, and then from there into the definition of `y`. A specification describes how the dataflow information is generated from intermediate code and how different pieces of dataflow information are combined. The dataflow engine then automatically computes the result which can then be used as the basis for various code optimizations. For example, knowing that the value `5` flows into `y`, the code above can be replaced with the simpler code:

```
int y = 5;
```

Much work has been done on each of these kinds of tool. In practice though only lexer generators and scanner generators are currently sufficiently automated to be widely used. You will therefore only use these kinds of tools in the homework assignments.