CPS323 Lecture: Programming Language Implementation

4/19/09

Objectives:

1. To overview the spectrum of options ranging from pure compilation to direct interpretation
2. To overview the structure of a compiler or interpreter
3. To overview the processes of tokenization and parsing
4. To introduce "optimization" strategies
5. To overview operations performed after compilation is complete (linking, loading)

Materials:

1. Projectable of Compiler Phases
2. Projectable of Pipe and Filter Architecture for a Compiler
3. Projectable of calculator.l and Scanner.java
4. Handout of Syntax Diagrams, parse tree, flex/bison grammar rules and Java interpreter code for a simple desk calculator
5. Projectable of parsing tables generated by bison for this grammar
6. Projectable of recursive descent parser for this grammar
7. Projectable of Java compileer for this grammar
8. Demonstration programs - interpreter for simple desk calculator produced from flex/bison rules (in calculator.l, calculator.y); interpreter produced from recursive descent parser (Scanner.java, Interpreter.java, InterpreterMain.java); "compiler" produced from recursive descent parser (Scanner.java, Compiler.java, CompilerMain.java).
9. Projectable of Figure 15.1

I. Introduction

A. Thus far in the course, we have focussed on programming language features with limited attention given to how these features are actually implemented. Now, we are shifting our focus to iimplementation.

B. Larger CS programs often offer full-blown courses - or even sequences of courses - relating to issues like compilation. Our approach here, though, will be very much a quick overview, since this is not really a "programming language implementation" course.

C. Nonetheless, some understanding of implementation issues is important for a variety of reasons.

1. Efficiency of implementation issues may influence what features are included or excluded from a programming language.

   a) Example: Many programming languages incorporate some sort of "switch" statement [e.g. FORTRAN computed `goto`, C/C++/Java `switch`; Ada `case`].

      Typically, it is required that the expression to be switched on be of an ordinal (discrete) type. We saw earlier in the course that such a construct has an efficient implementation using a jump table.

      OTOH, allowing the expression to be switched on to be of a non-discrete type would require explicit tests for each possible value (as in the LISP `case`), and is generally disallowed (LISP being the exception).

   b) Bjarne Stroustrup has this to say about the inclusion of virtual functions in C++ (the first C family language to offer dynamic dispatch)

      "The idea was borrowed from Simula and presented in a form that was intended to make a simple and efficient implementation easy" (*The Design and Evolution of C++* p. 72)

2. Appreciation of implementation issues may affect how one uses a particular feature

   Example: Knowledge of how two-dimensional arrays are stored in memory should influence which subscript is used for the outer loop and which for the inner loop when doing some operation on every element of an array.

II. Compilation and Interpretation

A. One sometimes hear's a particular programming language described as "a compiled language" or an "interpreted language". Examples:

   ASK

   The distinction is an important one, but not as simple as the use of these terms may sound.

1. There is, in fact, a spectrum of approaches.

2. It is better to think of compiled versus interpreted implementations of a language, rather than compiled versus interpreted languages, since virtually any language could, in principle, be implemented either way (though usually one alternative is a lot cleaner for a given language).

B. We begin with a basic definition: what does it mean to "interpret" a program?

ASK

(From CPS112 intro lecture): "We say that a computer system interprets a program when it carries out the actions specified by that program."

1. Thus, there is a sense in which all computer programs are interpreted. In part, the distinction between "compiled" and "interpreted" relates to the question of what does the interpretation - the hardware, or a software interpreter that is itself interpreted by the hardware? Thus, a more precise usage of the term "interpreted" (as that term is commonly used) would be "software interpreted."

2. A compiler is basically a translator that translates the source form of a program into a form that is more easily interpreted by hardware.

C. Perhaps a clearer way to look at the distinction is to look at two issues:

1. The user's perspective:

   a) One typically thinks of an implementation of a language as being a compiled implementation if the user must carry out an explicit compilation step between writing a program and running it.

      Examples from this course: FORTRAN, Ada

   b) OTOH, one typically thinks of an implementation of a language as being an interpreted implementation if the user can enter a statement and see the result of executing it immediately.

      Examples from this course: LISP, Prolog, Python, Ruby

2. The computer memory perspective

   a) One typically thinks of an implementation of a language as being a compiled implementation if there is a software component (the compiler) that is not (or at least does not need to be) present in memory when the program is running.

b) One typically thinks of an implementation of a language as being an interpreted implementation if the full implementation of the language needs to be present in memory when the program is running.

c) Notice that, under either model, there will be some components of the implementation present in memory when the program is running - e.g. a library of runtime support routines, or perhaps even a virtual machine implementation. The distinction is whether all the components of the implementation need to be present.

D. I mentioned earlier that there is really a spectrum of approaches - with the classic notions of compilation and interpretation being the ends of the spectrum.

1. Classically, compilers were designed to translate all the way to the machine language of some platform - which meant that a compiler would need to be both language and platform specific. Some newer language implementations are based on a compiler that generates code for a "virtual machine" which is then interpreted by a platform-specific virtual machine interpreter. (The target language is commonly referred to as "bytecodes").

   Examples?

   ASK

   a) A key advantage of this approach is portability - the same bytecodes can be run on any platform that has a virtual machine implementation. (c.f. Sun's "write once run anywhere" slogan for Java and the whole notion of an applet embedded in a web page.)

   b) There is a sense in which this approach could be regarded as lying between a purely compiled implementation and a purely interpreted implementation. However, this strategy is usually thought of as a compiled strategy because

      (1) The user must explicitly perform a compilation step before running a newly-written or edited program.

      (2) It is not necessary that the complete implementation be present in memory when the program is running - just the virtual machine. (However, in the case of Java at least, it does turn out that the complete compiler resides in the package `com.sun.tools.javac` in the standard library, which is how tools like Dr. Java can give the appearance of being Java intrpreters!)

c) Of course, the virtual machine approach is much less time-efficient than compilation all the way to machine language, since the virtual machine interpreter involves a lot of overhead (e.g. executing a single JVM bytecode requires several machine language instructions at least, and can require dozens.)

A common solution to this is the use of some sort of JIT (Just-in-time compiler)

(1) In the simplest case, the virtual machine invokes the JIT to compile a block of code (e.g. a method) to the native machine language of the platform before it is executed the first time. This, of course, means significant delay the first time a body of code is executed, but since the translated code is cached, code that is executed repeatedly needs to be translated just once.

(2) Some JIT's avoid this initial overhead by only translating code that is executed repeatedly. (Code that is just executed once is executed interpretively as on a system without a JIT).

(3) JIT technology can actually achieve superior performance to traditional compilation in a couple of ways

(a) A JIT can compile to the exact machine a program is running on, taking full advantage of features/characteristic of that particular machine, while traditional compilation must produce more generic code for a family of processors.

(b) A JIT only compiles code that is actually executed. (Typically, only a portion of the code of a program is executed during any particular run of the program.)

2. At the other end of the spectrum, language implementations that are basically interpreted often store a program internally in some intermediate form, rather than as raw source. This means that some of the most time-consuming steps in interpretation (e.g. parsing the source) are only done once.

a) Example: BASIC on RSTS/E (1980's - early 1990's) used this approach. The implementation supported a "compile" command which just dumped the intermediate form to a file (a very quick operation) for future execution. Most of the operating system utilities were written in BASIC and stored in this compiled form!

b) Again, this approach could be regarded as lying between a purely compiled implementation and a purely interpreted implementation. However, this strategy is usually thought of as an interpreted strategy because

(1) The user is not conscious of an explicit compilation step - the translation into intermediate form is done automatically when the source code is first read.

(2) The full implementation is still necessarily memory resident when the program is running.

c) It is also possible to utilize JIT technology with this approach - potentially achieving code (at least often-executed code) that is just as efficient as that which could be produced by pure compilation. (In fact, evidently the first use of a JIT was in such a context.)

E. While it is possible, in principle, to implement any language either way, there are characteristics of a programming language design that tend to make one approach or the other more natural.

1. It is commonly the case that languages that typically use a compiled implementation are staticly typed, and languages that typically use an interpreted implementation are dynamically typed. (This is certanly the case with the languages we have studied.)

Why is this so?

ASK

a) In an interpretive implementation, it is desirable for each statement to stand more or less on its own, with minimal dependence on other statements. But the meaning of a statement in a statically typed language is heavily dependent on declaration statements that proceed (or may even follow) it.

Example: In Java, `a + b` means something quite different if the variables are numbers than it does it at least one is a String.

b) A compiler can often generate more efficient code if it knows the data types of the variables involved in a statement.

Example: In most languages, the machine language instructions that translate a construct as simple as `a + b` are quite different if the variables are integers than when at least one is a real number. (Most modern computers have separate register sets as well as machine instructions for integer and real number arithmetic.)

2. Languages that allow a program to create and then execute code typically use interpretive implementations. Examples?

   ASK

   But note that this is not rigidly the case. For example, in Java, it is possible to generate and then execute code "on the fly" by taking advantage of the fact that the compiler actually is part of a package in the standard library. However, this is non-trivial!

III. The Structure of a Compiler or Interpreter

A. Individual compilers vary widely in their structure, but most can be analyzed in terms of six PHASES (not all of which are necessarily present in any given compiler.) The input to the first phases is the source program; the input of each subsequent phase is the output of the previous phase; the output of the last phase is the object program.

   PROJECT: 6 phase structure

   1. Lexical scanning

   2. Syntactic analysis (parsing)

   3. Semantic analysis

   4. Intermediate code generation [Note: text treats this as part of previous phase]

   5. Platform-independent code improvement

   6. Code generation, possibly including platform-dependent code improvement. [Note - some would classify platform-dependent code improvement as a seventh phase. The text classifies it as a sixth phase because it combines 3 & 4 above.]

      a) Sometimes, the code generated by this phase is actual machine code.

b) But often, the generated code is assembly language, which is then assembled as a separate step to produce the actual machine language. (This, for example, is the approach used by the gnu compiler collection in fact, by using the -S option on the compiler command line, one can stop the compilation with production of the assembly language which one can then view.)

Demo: Create the following via pico:

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

Then compile with `gcc -S` and show assembly language code

B. Sometimes, the phases are grouped into two categories: the "front end" (phases 1-4) and the "back end" (phase 6), with phase 5 either part of the front end or subsumed by phrase 6.

1. The former depends only on the language - not on the target platform.

2. The latter depends only on the target platform, but may be used for a variety of languages if a common intermediate language is used.

3. This leads to an interesting strategy for building compilers. Suppose one had 6 languages and 6 target platforms. In principle, one would need to write 36 compilers. Instead, assuming that one can design an appropriate intermediate language, one can construct 6 front ends - one for each language - and 6 back ends - one for each platform, which can be combined in various ways to yield the needed 36 compilers. (In fact, the gnu compiler collection does something essentially like this.)

C. The structure of an interpreter is similar to the first 3-4 phases of a compiler, but code to actually interpret the output of semantic analysis or intermediate code generation replaces the last few phases. Because of this similarity, we will focus on the structure of a compiler here.

D. Associated with these six phases are two other major components of the compiler, that are utilized by the various phases.

1. The symbol table, used to record information about identifiers and their declarations.

   a) The symbol table may be used by the lexical analysis phase to discriminate between reserved words and identifiers, if reserved words are entered as such in the symbol table. That is, the lexical analyzer may encounter a string that could be either a reserved word or an identifier, and then can ask the symbol table which it is. (If it's not there, of course, the assumption is that its an identifier.)

   b) The parser uses the symbol table to record various attributes of an identifier when it is declared (or perhaps just the fact that it is a variable or function name in the case of a dynamically-typed language.)

   c) Semantic analysis may use the symbol table to retrieve attributes of an identifier in order to analyze a form - e.g. what does `a + b` mean?

   d) Intermediate code generation also uses declaration information recorded in the symbol table - e..g. accessing a local variable, a lexical variable, a global variable, or a parameter calls for quite different code.

2. The error handler, which is called whenever an error is detected during one of the phases.

E. The 6 phases are generally organized into one or more PASSES. Each pass involves reading a file and writing a file.

1. Example: A 6 pass compiler would do all lexical analysis first, producing a temporary file of tokens. The parser would read this file and produce a second temporary file containing a parse tree. On the third pass, the semantic analysis phase would read this and produce a third temporary file containing intermediate code ... (Actually, it would be very unlikely for any compiler to implement every phase as a separate pass.)

2. On the other hand, in a 1 pass compiler, all steps would be handled for each token or group of related tokens as they are encountered without generating any intermediate files. The compiler might be organized as a collection of coroutines or even as a set of programs connected together via pipes, using the pipe and filter architecture.

   PROJECT: Pipe and filter architecture for a compiler

3. Some languages require at least two passes to compile due to forward reference issues.

Example: you may have noticed, when working with Java, that sometimes you fix a syntax error and all of a sudden get a lot more. This happens because certain kinds of errors cause the compiler to abort after the first pass, so syntax errors that are detected on the second pass would not be discovered until the pass one problem is fixed.

4. On the other hand, some languages (e.g. Pascal) are designed to facilitate one pass compilation.

   a) This was considered important for Pascal because of its intended use as a teaching language, to minimize the time spent on compilatiion.

   b) To support this, Pascal requires all identifiers to be declared before they are used, and provides two forms of "forward" declaration syntax to allow for mutual recursion among procedures and for recursive data types..

      (Note that Java requires declaration before use in many cases, but not all; in particular, a method in a class may reference instance variables or other methods that are declared later in the class definition.)

5. Dividing a compiler into separate passes may be done for the sake of modularity, even if the language being compiled does not require it.

F. The various phases communicate with each other by means of appropriate data structures:

   SHOW PASS STRUCTURE PROJECTABLE AGAIN, LOOKING AT DATA STRUCTURES

   1. Input to phase 1: source program (sequence of characters);

   2. Phase 1 -> 2: a stream of tokens, which can be internally represented as records having various attributes that classify them.

   3. Phase 2 -> 3: a concrete parse tree (or more often subtrees of it).

   4. Phase 3 -> 4: an abstract syntax tree (derived from parse tree, with semantic anotations)

   5. Phase 4 -> 5, 6: intermediate code (a machine-independent representation of the executable code)

   6. Output from phase 5: (virtual) machine language code

G. The compiler structure and data structures we have outlined is not rigid; actual compilers may combine phases and/or not actually construct all the data structures listed.

1. Often, semantic analysis is combined with parsing, by augmenting the grammar with action rules that specify what action to be taken when a particular syntactic construct is recognized. (The examples we will use shortly do this.)

2. A very simple compiler might be structured as just two components: a lexical scanner, and a component which combines parsing, semantic analysis, and (final) code generation. In this case, here would be no separate intermediate code generation or code improvement phases, and the parse tree would not actually be constructed. (We will look at an example that is structured this way later.)

3. However, the separation of the lexical scanning phase from subsequent phases is almost universal. Mixing lexical scanning with parsing would greatly complicate life!

H. Each phase of a compiler is a signficant subject in its own right.

1. A classic description of compiler structure is:

   Aho, Sethi, and Ullman - Compilers: Principles, Techniques, and Tools (Addison Wesley, 1986) - Also known as the "Dragon Book"

2. For our purposes, we will focus on two phases in particular: the parsing phase and the code improvement phase.

IV. Tokenizing and Parsing

A. In our overview of the structure of a compiler (or interpreter)., we noted that the first two phases are called lexical scanning and parsing. A program is, in one sense, just a big character string! The task of lexical scanning is to break it up into basic units of meaning called tokens - e.g. identifiers, reserved words, numbers, operator symbols, punctuation, etc.

1. The grammar of the tokens of a language is typically specified using regular expressions - e.g. the C/C++/Java token type identifier might be specified by

   identifier -> ([A-Z] | [a-z] | $ | _)([A-Z] | [a-z] [0-9] | $ | _)*

(Note that this is also the syntax for a reserved word. The distinction between the two is commonly made by a table lookup to see whether a given "identifier" is actually defined as a reserved word).

2. There exist automated tools (e.g. lex or flex on Unix/Linux systems) which will create a lexical scanner given a token grammar expressed as regular expressions.

   Example: PROJECT flex scanner for simple desk calculator - calculator.l

3. It is also possible to write a lexical scanner by hand.

   Example: PROJECT Java scanner for simple desk calculator - Scanner.java

B. The normal way to construct a parser is to start with a well-defined formal grammar for the language to be parsed - typically a context-free grammar, with non-context-free issues (like the "declare before use" rule for identifiers) handled in an ad-hoc way.

1. Indeed, for some classes of grammars (including many of real interest), it is possible to construct a parser automatically given a description of the grammar in suitable form. (For example, UNIX includes a utility yacc and Linux includes yacc plus the gnu version bison that generate a parser from a description of a suitable grammar.) (The name yacc stands for "yet another compiler compiler", and bison is a play on words on yacc (yak))

2. That is, these tools are translators whose source language is the description of a language, and whose object is a parser for the language (written in C)

3. To give us a concrete example, we are going to use a very simple programming language that has just one statement: `print` *`expression`*, where *`expression`* uses the syntax we used in the Ada project.

   a) HANDOUT: Calculator example handout. Discuss Syntax graphs and parse tree

   b) DEMO: Interpreter generated from flex/bison grammars on handout

C. Parsing has been extensively studied - e.g. in the class two volume work on the theory of compilation by Aho and Ullman, one is devoted to parsing.

1. One important theoretical issue is the relationship between grammars and appropriate parsing methods.

   a) Tools like yacc and bison, for example, impose certain requirements on the grammar.

   b) There are other parser construction techniques that impose different requirements.

   c) Considerations of what strategy may be used for parsing the language can help shape the grammar - indeed, there are some features that are part of programming language grammars precisely to facilitate the construction of a parser.

      Example: Many programming languages require some symbol to occur between the condition part of an if statement and the "then" statement - e.g.) in C family languages; `then` in Pascal/Ada family languages, `->` in Prolog. This facilitates parsing an if statement, by providing a clear line between the condition and then parts whereas otherwise it would not always be obvious where one ended and the next began.

2. Some nomenclature regarding parsing methods:

   a) There are two general approaches to parsing: bottom-up and top-down - named on the basis of how they generate the parse tree:

      (1) Note that in a parse tree the leaves are symbols in the language itself (terminal symbols) - such as keywords, identifiers etc.

      (2) The internal nodes are non-terminal symbols - names for syntactic classes - such as expression, term

      (3) The root of the tree is a non-terminal symbol that classifies the entire structure - e.g. a program.

      (4) A bottom-up method constructs the tree from the leaves to the root by saying, in essence "what I see in the source program is _____. This is the beginning of a _____." When it builds a complete entity, such as an expression, it then tries to fit it into a larger context. Shift-reduce parsing is a bottom up approach that is used by parser generators such as yacc. A shift-reduce parser uses a stack to store partially-completed subtrees of the parse tree.

Ex: For our example grammar and the sample input `print - 2 + 3 * 4`, a bottom-up approach would proceed as follows: (Draw stack on board)

- `print`: Shift onto stack
- -: Shift onto stack
- integer 2: Shift onto stack
- Reduce integer 2 to factor
- Reduce - factor to factor
- Reduce factor to term
- Reduce term to expression
- +: Shift onto stack
- integer 3: Shift onto stack
- Reduce integer 3 to factor
- Reduce factor to term
- *: Shift onto stack
- integer 4: Shift onto stack
- Reduce integer 4 to factor
- Reduce term * factor to term
- Reduce expression + term to expression
- ;: Shift onto stack
- Reduce print expression ; to statement

A shift reduce parser uses the incoming symbol to decide whether to reduce at any point in the parse. For example, we reduced the factor -2 to term and then to expression because the incoming token was +, which can come after a expression but not after a factor or term. But in the case of the integer 3, though we reduced it to factor and then term we did not reduce it to expression because the incoming token * comes after a factor.

The heart of what bison does with a grammar is to produce a set of tables that guide the parser in deciding whether to shift or reduce - and how far to reduce - based on the incoming symbol and what's on the stack (which is encapsulated in a state.)

PROJECT: tables constructed by bison

(5) A top-down method constructs the tree from the root to the leaves by saying, in essence "we want to build a _____. To do so, we first need to find a _____" - and then it looks for the desired building block in the input. If the building block is itself a non-terminal, then it says "to find a _____ I must first find a _____ etc." (Recursive descent parsing - which we will look at shortly - is a top-down method)

Ex: For our example grammar and the sample input
"`print - 2 + 3 * 4 ;`", a top-down approach would proceed as follows:

- We are looking for a statement.
- A statement must begin with `print`. The input begins this way.
- Now we are looking for an expression.
- An expression begins with a term, so we are looking for a term.
- A term begins with a factor, so we are looking for a factor.
- One way a factor can begin is with `-`. The next token in the input is this.
- Now we are looking for another factor.
- Another way a factor can begin is with an integer. The next token in the input is `2`. This factor, together with the "-" we saw earlier, constitutes a factor.

...

Error messages of the form "____ expected" may follow from the structure of a top-down parser, since an error is reported whenever the parser does not see what it is looking for. For example, a top-down parser for our example grammar might output the message "print expected" if confronted with the input text
"`- 2 + 3 * 4`".

PROJECT recursive descent parser created for this grammar: Interpreter.java (also in handout)

b) Parsers are sometimes classifiied by using the notation LL(k), LR(k), RL(k), or RR(k). [Though RL and RR are hardly ever used.].

(1) The first letter indicates the direction in which the parser reads the source text - L meaning "left to right". So an LL or LR parser is one that reads the source text left to right. (Parsers that read the text right to left are really of only theoretical interest.)

(2) The second letter indicates the direction in which the parser constructs the parse tree. An LL parser constructs the parse tree left to right - and LR parser constructs it right to left. Our first example was actually a right to left parse - due to the use of a stack, the complete parse tree was discovered right to left. Our second example (recursive descent) produced a left-to-right parse.

(3) k is an integer constant specifying the number of tokens of lookahead that is necessary for the parser. The most common value is 1 - the parser needs to lookahead at most one token in the input (Lookahead of 2 or more tokens leads to an unwieldy parser, and only the simplest of grammars require no lookahead. Our two examples were, respectively, LR(1) and LL(1).)

(4) Grammars can also be classified as LL(k) etc, based on what kind of parser could handle the grammar. Of course, it is often the case that more than one classification applies to a grammar - for example, our example "bison" grammar is LR(1), while our Syntax Graph grammar is LL(1).

(5) The grammars that are easiest to parse are those which are suitable for an LL(1) or an LR(1) parse - or both. There are, of course, languages which cannot be parsed in these ways; but programming language designers generally try to produce a grammar that is either LL(1) or LR(1) or both - or at least close enough that the rough spots can be dealt with by variious techniques.

D. Although semantic analysis is a separate phase of a compiler, it is often in practice closely coupled to the parsing phase. With each routine in the parser that recognizes a given syntactic construct, we can associate action routines that deal with the associated semantics - e.g. placing an identifier in the symbol table when we recognize a declaration.

1. Note how this has been done in the calculator examples - in both cases, the parser actually includes the code to do the interpretation.

2. It is also possible to augment a parser with the remainder of the code to do compilation.

   SHOW, DEMO Compiler.java

V. "Optimization"

A. One important consideration in designing or choosing a compiler is the extent to which the compiler produces code that is as good as that which might be produced by a skilled assembly language programmer.

1. Here "goodness" may be measured in terms of total code size, or in terms of execution speed, or both. (Note that code size and speed sometimes conflict; but often saving one saves the other and vice versa.)

2. Compilers that perform simple line by line translations of the source code tend not to do very well on this score. For example, consider the following series of C++ statements (where the variables have previously been declared as ints)

```
sum1 = x0 + x1;
sum2 = sum1 + x2;
sum3 = sum2 + x3;
sum = sum3 + x4;
```

When this is compiled with gcc on an Intel platform without optimization, the resultant code looks like this:

Load x0 into register edx
Load x1 into register eax
Add register edx, register eax into register edx
Store register edx into sum1
Load sum1 into register edx
Load x2 into register eax
Add register edx, register eax into register edx
Store register edx into sum2
Load sum2 into register edx
Load x3 into register eax
Add register edx, register eax into register edx
Store register edx into sum3
Load sum3 into register edx
Load x4 into register eax
Add register edx, register eax into register edx
Store register edx into sum

That is, three times the generated code stores a register into a variable and then (unnecessarily) loads the same variable into the same register

3. However, so-called "optimizing" compilers can produce code that rivals the best that any human translator might produce. (The term "optimizing" is actually a misnomer, since one cannot really define what "optimal" code would be. Nonetheless, this is the term that is used.) For example, when gcc compiled the above with optimization turned on, the redundant loads were eliminated.

4. Optimizing compilers can also correct for certain kinds of carelessness on the part of the programmer - e.g. creating variables that are never used, doing computations inside a loop that could be done outside it, etc.

5. In general, optimizing compilers trade extra effort at compile time for reduced object program size and/or faster execution at run-time.

   a) Some optimizing compilers allow the user to specify whether time or space should be the primary goal in compilation.

   b) Many optimizing compilers allow the user to disable optimization, to get faster compilations during program development. (Since the resultant object program is only going to be executed one or a few times before changes are made, optimization is not important.)

   c) One subtle problem that can arise is that sometimes the code generated by an optimizing compiler behaves differently from the unoptimized code due to semantic ambiguities in the language.

      Example: Recall that Pascal does not specify the order in which tests connected by "and" or "or" are done. Consider the following code:

      ```
      while (p <> nil) and (p^.info < wanted) do ...
      ```

      - The programmer obviously wants the `p <> nil` check to be done first; otherwise a "null pointer" exception would occur

      - It may be that the optimizing version handles this differently from the non-optimizing one. (Actual experience with the VAX Pascal compiler!)

   d) Also, since an optimizer may significantly rearrange the code (without changing its meaning), one usually wants to turn off optimization when one intends to use a symbolic debugger on the code.

B. Our goal in this portion of the lecture is to look at the KINDS of possible optimizations a compiler might do, but NOT HOW the compiler figures them out. (The book spends a lot of time on this, but we will not given the amount of time left in the course!)

C. Optimizations may be classified into:

   1. Platform-independent optimizations - i.e. all compilers for a given language would do the same thing.

   2. Machine-dependent optimizations - i.e. those that take advantage of the register set or special instructions of a given machine.

3. In our 6-phase model of a compiler, platform-independent code improvement is phase 5 of the process; platform-dependent code improvement is either considered part of phase 6 or a separate phase 7.

D. Optimizations may also be classified as LOCAL or GLOBAL or INTERPROCEDURAL.

1. For purposes of optimization, the code can be thought of as broken into BASIC BLOCKS. A basic block is a sequence of statements that contain no transfers of control internally or in or out. (Hence, transfer of control instructions themselves are not part of any basic block.)

a) Example: consider the following code:

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

An optimizing compiler might perform this as if it were written:

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

b) However, suppose there were a label on the second statement:

```
        a = b+c;
foo:    d = b+c;
```

Now, this transformation is impossible. If control reaches the second statement via a `goto foo`, then we cannot rely on `a` having been set equal to `b+c`.

2. Basic blocks must be defined in terms of the structure of the code that will ultimately be produced:

Example:
```
if (a > b)
        while (c > 0)
        {
                a = a + b;
                c = c - b;
        }
```

Becomes something like

```
        compare A to B
        branch if <= to L2
L1:     compare C to 0
        branch if <= to L2
        add a to b
        subtract b from c
        branch unconditional to L1
L2:
```

This code contains three basic blocks: the first comparison (by itself), the comparison to zero (by itself), and the add and subtract (together).

3. Local optimizations are carried out within the scope of a single basic block. Global optimizations transcend the basic blocks, and depend on analysis of the control flow of a single routine - i.e. consideration of the various paths by which a given block can be entered. As we shall see, it may even move code from one basic block to another. Interprocedural optimizations consider control flow between routines - i.e. what routines call what routines.

   Our examples will pertain primaily to either local or global optimization. Interprocedural optimization is much more complex and still very much a research area.

4. Some, but not all, optiimizations may be performed using a technique called peephole optimization. Such optimizations look at a small number (2-3) of successive instructions at a time.

   For example, the elimination of redundant load operations in the example we looked at earlier could be done by peephold optimiztoin.

5. More sophisticated optimizations are done by looking at larger portions of the abstract syntax tree. In fact, though we have treated optimization as a phase in the compilation process, it is not uncommon for optimation to be done via a series of sub-phases, and some possibiities may need to be explored more than once because some optimizations make others possible.

   PROJECT Figure 15.1 from book

E. Some Typical Machine-Independent Optimizations

1. Avoiding redundant store/load operations

a) As we saw earlier, the following sort of sequence of operations arises frequently in doing a straight translation of source code to machine code:

```
store     some-reg, some-place
load      the-same-place, some-reg
```

Of course, if both instructions lie in the same basic block, then there is no way control can reach the load except by going through the store so the load is not needed and can be eliminated.

b) This is a local optimization. We consider it machine independent since it is typically done on the intermediate code, which makes use of virtual registers to stand in for the architectural registers that will actually be used in the final code.

2. Constant Folding

a) One simple optimization is to replace arithmetic operations involving constants at by a single constant calculated at compile time.

Example:

```
a = sin(theta * 3.14159 / 180);
```

Can be optimized to:

```
a = sin(theta * 0.01745);
```

b) This can be done locally. It reduces both execution time and code size.

3. Constant Propagation

a) Constant propagation generates new opportunities to do constant folding on the basis of knowledge that a particular variable must have a particular constant value at a particular point in the code.

Example:

```
a = 3;
b = a * 7;
```

Can be optimized to:

```
a = 3;
b = 21;
```

b) This can be done both locally and globally.

(1) The above example is local, of course.

(2) Doing this globally requires the compiler to anlayze the flow of the routine, leading to a determination of what value(s) may be possible for a given variable at a given time

Example:

```
a = 3;
for (int c = 0; c < 10; c ++)
     cout << c << endl;
b = a * 7;
```

The assignment to `b` can be replaced by `21` as before, since even though the assigments to a and `b` occur in different basic blocks, the intervening code can have no affect on the value of `a`. (Assuming, of course, that `a` and `c` are not aliases - something which the compiler would have to check for!)

(3) Interprocedural analysis of routine calls may lead to discovering an opportunity for propagating constants from one routine to another that it calls - provided the compiler can know all of the possible call points for a routine (e.g. if the routine is local or private).

c) It saves both execution time and code size.

4. Strength Reduction

a) In strength reduction, we replace an "expensive" operation by a cheaper one.

Examples:

(1) 2*a becomes a + a or a << 1

(2) a**2 becomes a*a

b) This saves execution time, and may save code space if the "expensive" operation involves more code. It can be done locally.

c) A variant of this makes use of algebraic identities to avoid unnecessary computations - e.g. since x + 0 = x, and an addition operation in which one addend is known to be 0 can be eliminated.

5. Elimination of redundant expression evaluation.

   a) Often times, the same expression will appear more than once in a context where it has to have the same value each time it is evaluated. In this case, it can be evaluated once and the result saved for use elsewhere.

      Example:
      ```
      y := (x + y + 1) * (x + y + 1);
      ```

      Can be optimized to:
      ```
      temp := (x + y + 1);
      y = temp * temp;
      ```

      (This can be especially efficient if `temp` is a register, which is likely to be the case for the evaluation of the expression anyway.)

   b) This optimization saves both time and space. It can be performed on both the local and global level.

6. Elimination of useless assignments and dead variables.

   a) An assignment is said to be USELESS if the value it assigns is never used.

      Example:
      ```
      { int i = 3;
        cout << "Hello, world" << endl;
      }
      ```

      The compiler could eliminate the assignment and - in this case - the variable itself from the object code: no storage, no initialization, etc.

   b) A variable is said to be DEAD if its value will not be used again.

      Example:
      ```
      { int i;
        for (i = 1; i < n; i ++)
            cout << i << endl;
        -- code not involving i
      }
      ```

      Here, i is dead once the loop is exited. A dead variable may have its storage allocation re-used for some other purpose. This is particularly helpful if the variable was stored in a register; that register now becomes free for some other use.

c) Useless assignments and dead variables must usually discovered by global analysis of the routine, of course. (This same sort of analysis can also discover certain programming errors, such as the use of a variable that has not been initialized, or a path out of a function that doesn't return a value.)

7. Frequency Reduction

a) One important class of optimizations focuses on reducing the number of times a certain statement is executed, usually in conjunction with a loop.

b) Basically, we move computations inside a loop whose result is invariant to before the loop.

Example

```
for (int i = 0; i < 100; i ++
     x[i] = x[i] + sqrt(z);
```

Can be optimized to:

```
double temp = sqrt[z];
for (int i = 0; i < 100; i ++
     x[i] = x[i] + temp;
```

c) This is a global optimization, because it requires moving code from one basic block (the loop body) to another. Note that this improves execution time, at the possible expense of using a bit more space for a temporary variable.

8. Code Hoisting

a) This is a similar-appearing optimization that saves space, but not time. (It doesn't cost time, though.)

b) Basically, if identical computations are done on both sides of a decision structure, if possible, we move them out of the decision structure, and do them outside the decision structure.

Example:

```
if (a > b)
{
     d = 1;
     c = a;
}
else
{
     d = 1;
     c = b;
}
```

Can be optimized to:

```
d = 1;
if (a > b)
     c = a;
else
     c = b;
```

c) Again, this is a global optimization that moves code from one block to another.

d) Of course, there will be cases in which an optimization like this is not possible:

Example:

```
if (a + d > b)
{
     d = 1;
     c = a;
}
else
{
     d = 1;
     c = b;
}
```

9. Loop unrolling.

a) A "for" loop that is done a constant number of times may be replaced by that number of repetitions of the loop body, with the loop control variable plugged in as a constant in each.

Example:

```
for (int i = 0; i < 4; i ++)
     cout << i << endl;
```

Can be optimized to:

```
cout << 0 << endl;
cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;
```

b) This saves execution time, but may require additional space.

(1) I said "may" because sometimes the overhead of the loop requires more code space than the repetition of the body.

(2) Of course, for a loop done enough times, unrolling will necessarily require additional space. A given compiler will have to incorporate some decision as to when the space cost exceeds the benefits of the time saved. (We would probably not want to unroll this loop if it were being done 100 times.)

c) Note, too, that it would be hard to unroll a loop whose number of iterations is not a constant, unless we could learn something about the value by constant propagation.

10. Inlining procedures.

a) We saw that, in some languages, the programmer has the option of specifying that a certain procedure is to be expanded inline whenever it is called, instead of being coded as a closed subroutine.

b) In languages that don't provide this facility (and possibly even in those that do) an optimizing compiler may choose to make a certain procedure or function inline code.

(1) This is frequently advantageous if it is called only from one place, saving both time and space.

(2) It may be advantageous for procedures called many times if the calling sequence overhead (e.g. passing parameters) is of comparable length to the body of the procedure.

Example:

```
boolean isempty(Stack s);
{
     return s == null;
}
```

The inline code consists of a simple test to see if s is zero (the typical representation for null) - 1 or 2 instructions on many machines. OTOH, the calling sequence involves putting the parameter in the parameter list and calling the subroutine - a minimum of 2 instructions just to do the call - and the subroutine itself not only involves the test but also the code to return to the caller, which necessarily involves more code than the inline!

F. Machine Dependent Optimizations

1. While these optimizations are obviously platform-specific, there are a few kinds of issues that show up across platform lines.

2. Use of the register set

   a) Most CPU's have some number of more-or-less general purpose registers available for use as accumulators, temporary storage, index registers, etc. Since accessing data in registers is much faster than accessing data in memory, an optimizing compiler will try to use these in the best way possible.

      (1) Variables which are accessed frequently - especially in the body of a loop - may be stored in a register, rather than in memory. (A prime candidate here is the index variable of a for loop.)

      (2) Temporary variables created by frequency reduction or code hoisting are also good candidates for being placed in registers.

   b) A good compiler will make use of information about deadness of a variable stored in a register to allow the same register to be re-used for some other variable later in the same section of code.

   c) Register allocation must be done carefully, though. There are times when putting a variable in a register may do more harm than good! For example, generally, when a procedure is entered, any registers it uses must have their current contents saved on the stack, to be restored on procedure exit, to prevent interference with the caller's use of them. This creates extra memory references that might take more time than would be used if the item in question were not put in a register at all!

   d) Good register allocation turns out to be a key factor in producing good code. (Maybe THE key feature.)

   e) The C and C++ languages have a storage class `register` (which is syntactically like `static` or `extern`), which provides a hint to the compiler that a particular variable is heavily used and therefore is a good candidate for putting in a register. However, modern compilers treat this only as a hint, and may in fact ignore it, since register assignment algorithms have progressed to the place that they typically do much better than a human can.)

3. Use of special machine instructions (idioms).

a) Many CPU's have machine instructions that provide a more efficient way of doing special cases of more general operations.

A simple example: The Intel architecture includes a variant of the add instruction that allows one of the addends to appear in the instruction itself (an immediate instruction). Thus, if x is in register eax, then,

x = x + 3

could be translated into the single machine instruction

add eax, 3

The Intel architecture also includes the instructions inc (increment) and dec (decrement), which are shorter since they only need to specify one operand. Thus, the idiomatic translation of

x = x + 1

would be

inc eax

b) On some architectures, there are also cases where a single machine instruction might perform several operations that are useful for particular purposes. For example, on the VAX architecture the following loop:

```
int [] x;
...
for (int i = 0; i < x.length; i ++)
    x[i] = 0;
```

Could be translated by the following

```
    movl length of x, r0
    moval x, r1   ; Base address of array to r1
L: clrl (r1)+
    sob r0, L
```

where the loop body consists of just two instructions!

VI. What Happens After Compilation: Linking and Loading

A. In order to actually execute a compiled program, its contents (typically stored as a disk file) must be copied into memory.

1. This process is commonly referred to as loading.

2. In the case of an implementation that uses software interpretation, this is not relevant since the source program is loaded into memory before the process of parsing etc. begins. So the remainder of the discussion here only applies to compiled implementations.

B. The compilation process we have just discussed usually does not result in a form of the program that is ready to load and run.

1. Many times, a program consists of multiple, independently-compiled modules.

2. Almost all programs rely on various standard library routines - either explicitly refrenced by the code or inserted by the compiler to perform certain more complex operations.

a) Example: instrinsic functions in FORTRAN and many langauges since

b) Example: if `x` is an `int`, the Java compiler translates an operation like `"x = " + x` as if it were written

```
"x  = ".append(Integer.toString(x))
```

3. The step whereby independly-compiled modules and library routines are compiled to produce a single executable program is called linking. Unfortunately, though, sometimes the term "loading" is (imprecisely and confusingly) used for this step.

Example: On Unix systems, compilers like `gfortran` or `gcc` translate a source module with a name like `foo.c` into an object module with the name `foo.o`. A separate program (which is used for many languages) is then invoked to combine all the ".o" files, together with routines from library files, into a single executable file known as `a.out` unless one specifies a different name on the command line.

a) The linker that is invoked is known as `ld` . (The name appears to be an abbreviation for "load"- though the man page makes it clear that the program is really a linker.

b) This process is usually transparent to the user, because the compiler command (e.g. gfortran or gcc) ordinarily invokes both the compiler and the linker, and then deletes the ".o" files. (However, one can suppress the linking step and preserve the ".o" files by including the -c switch on the command line).

Demo:

Given the following do-nothing function contained in a file foo.c

```
void foo() { }
```

`gcc foo.c` : ld reports an error because there is no main program.

gcc -c foo.c stops at foo.o

C. In the approach that was standard since the earliest days of higher level languages, the executable file produced by linking contained all the executable code required for the program. This, however, has several problems:

1. There are certain library routines that are used by almost every program (e.g. the startup code that is executed before main() is actually invoked, basic IO, etc). Under this approach, each executable file contains a copy of all this code (and often other code from the same library module),

2. If multiple programs are being run at the same time, then multiple copies of this common code take up space in memory.

3. If any library code is updated, all programs that use it need to be relinked to take advantage of the updated code (normally not a problem if the update adds a new feature that the program doesn't use anyhow, but potentially a serious problem if the update is a bug fix or adapts to a change in external protocols.)

D. Today, however, almost all computers use some form of dynamic linkilng, in which a single copy of key code is resident in memory and is shared by all programs that use it. Instead of copying this code into the executable image, the linking step creates a reference to this shared code which is resovled when the program is loaded.

(This is the origin of so-called "dll conflicts" on Windows platforms, because program installers are allowed to install the needed dynamic link libraries, which can mean that the installation of a program that was linked against an earlier version of a dll may cause the replacement of the newer version needed by some other program. For more of this, lookup "dll hell" on Wikipedia!

E. Languages like Java carry this concept further.

1. The analogue to linking in the Java world is creating a .jar (Java archive) file that contains all the classes created for the program (but no library classes).

2. When a Java program is run, only the class containing the main program is actually loaded into memory. Other classes are loaded into memory only when referred to by the running program via a reference to a class field or method, or by invoking a constructor.

3. The classes comprising the Java class library reside in a single (rather large) .jar file installed as part of the JRE; they, too, are only loaded when needed. Of course, installing a new version of the class library results in all programs that run subsequently using the new versions of the library classes. (Which implies that new versions of the library must maintain backward compatibility with previous versions, though it is possible to declare a class or method deprecated, meaning it will eventually be totally removed, but not until several versions later.)

4. On Windows platforms, .NET does something similar.

F. Some language implementations have offered a "load and go" compiler which translates, links, and loads a program in a single operation. This is used when it is only intended to run the resultant program once - as might be the case during initial development or for student projects.