CS426
Compiler Construction

Fall 2006

David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
0. Course organization
Instructor:  David A. Padua  
4227 SC, 333-4223  
Office Hours: By appointment  
padua@uiuc.edu  
http://polaris.cs.uiuc.edu/~padua/cs426

TA:  James Brodman  
4301 SC, 333-6578  
Office hours:  
brodman@gmail.com

Time & Place:  12:30 PM - 01:45 PM  TR 1103 SC

Credit:  3 or 4 hours

© 2007  Not Yet Published  
A. Aho, M. Lam, R. Sethi, and J. Ullman

Evaluation:  Midterm (Oct. 12)  25%  
Final exam (Dec. 14 [7-10pm])  40%  
Term project (Dec. 14)  35%  
Term paper for 4th hour (Dec. 14)
1. Introduction
What is a compiler?

- Compilers are programming language translators.

- Accept *source language*.

- Translate into *target language*.

- Numerous source and target languages, but fairly general techniques have been developed, and some reuse of software and automatic generation of compiler modules is possible.

- Although much progress has been made since the first commercial optimizing compiler (Fortran I), compiler development is still a challenging task.
Importance of compilers

- Compilers facilitate programming by
  - Presenting a high-level interface
  - Enabling portability

- In some domains, the acceptance of a programming language is influenced by the availability of effective compilers. That is, compilers that generate fast target code.
• For that reason, an important objective of compilers has always been code optimization:

“It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger.”

John Backus
Fortran I, II and III
• Compiler optimization attempts to completely liberate programmers from machine concerns and thus enable portability without loss of performance.

• Goal is for programmers to focus on the development of clean (easy to understand and debug) programs and ignore machine details.

• Today’s compilers are not always 100% successful at these tasks and some human involvement is usually required to obtain highly efficient code.
The analysis-synthesis model of compilation

The two mains tasks of compilers are:

- **analysis**
  - source program

- **synthesis**
  - target program

**Intermediate representation**
• Analysis
  -- breaks up source program into constituent parts.
  -- gathers information implicit in the source code and detects errors.
  -- creates an intermediate representation.
  -- the textbook gives in Chapter 1 a narrow definition of analysis as applying only to the source code. As we will see later, analysis is often also performed on the internal representation.

• Synthesis
  -- Generate the target code by manipulating and transforming the source code.
  -- Translates and often also optimizes
Syntax analysis

- The first type of analysis that must be performed in syntax analysis. It produces parse trees and abstract syntax trees whose shape follow the structure of the source program.

- For example, even a simple the statement like

  position = initial + rate * 60

has an structure determined by operator precedence. For example, its abstract syntax tree has the following form:
The abstract syntax tree is the culmination of a process that involves **lexical analysis or scanning** and parsing.

Lexical scanners read sequence of characters and organizes them into **tokens**. These are elementary components of programs such as keywords (or reserved words), variables, special characters, numerical constants, etc.

Parsing builds a parse tree (but it may not save it) from the **tokenized** input based on a grammar that describes the language accepted by the compiler. Parsing detects syntax errors.
A simple grammar for assignment statements is:

$$\begin{align*}
A & \rightarrow \text{id} = E \\
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow T \ast F \\
T & \rightarrow F \\
F & \rightarrow (E) \\
F & \rightarrow \text{id} \\
F & \rightarrow \text{number}
\end{align*}$$

The parse tree is then:

```
A
  =
  E
    +
    T
      *
      F
        id
        id
        initial
        rate
        number
        60
```
Semantic analysis

- Gathers information for the subsequent code generation phase
- Check code for semantic errors
  - Type of objects
  - Undefined variables
  - Parameters to intrinsic functions
  - Java compiler / VM do extensive semantic analysis
Overall organization of a compiler

Front end

- Lexical scanner
- Parser
- Semantic analyzer

High level representation (abstract syntax tree)

- High-level optimizations

Back end

- Intermediate representation
- Low level optimizations
- Machine code generator

Target program
The context of a compiler

skeletal source program

preprocessor

source program

compiler

target assembly
language program

assembler

relocatable machine code

loader/link editor

absolute machine code

target machine
(hardware interpreter)

Book

light gray boxes are not always present

Course project

source program

compiler

target JVM assembly
language program

assembler

.class file

Java Virtual Machine Interpreter
Compiler Optimizations

First, a note about the word *optimization*.

- It is a misnomer since there is no guarantee of optimality. Many problems are NP complete or undecidable.

- We could call the operation code improvement, but we should follow tradition.

- In any case, real implementations not always improve the code since compiler transformations are not guaranteed to improve the performance of the generated code.
A classification of compiler optimizations

By the scope

- **Peephole optimizations.** A local inspection of the code to identify and modify inefficient sequence of instructions.

- **Intraprocedural.** Transform the body of a procedure or method using information from the procedure itself.

- **Interprocedural.** Uses information from several procedures to transform the program. Because of separate compilation this type of optimization is infrequently applied to complete programs.
By the time of application

- **Static.** At compile-time
  - Source-to-source
  - Low-level optimizations

- **Dynamic.** At execution time.

By the source of the information

- **Code only**
- **Code plus user assertions**
- **Code plus profile information.**
Which optimizations to include?

- The optimizations must be effective across the broad range of programs typically encountered.

- Also important is the time it takes to apply the optimization. A slow compiler is not desirable (and for some transformations the compiler can become very slow).
Order and repetition of optimizations

A possible order of optimizations, shown in the figure below, is from S. Muchnick’s book “Advanced compiler design implementation”.

Two quotes from that book:

“One can easily invent examples to show that no order can be optimal for all programs.”

“It is easy to invent programs that will benefit from any number of repetitions of a sequence of optimizing transformations. While such examples can be constructed, it is important to note that they occur very rarely in practice. It is usually sufficient to apply the transformations that make up an optimizer once, or at most twice to get all or almost all the benefit one is likely to derive from them.”
In-line expansion
Leaf-routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimizations and conditional moves
Dead-code elimination
Software pipelining, with loop unrolling, variable expansion, register renaming, and hierarchical reduction
Basic-block and branch scheduling 1
Register allocation by graph coloring
Basic-block and branch scheduling 2
Intraprocedural 1-cache optimization
Instruction prefetching
Data prefetching
Branch prediction

Interprocedural register allocation
Aggregation of global references
Interprocedural 1-cache optimization

(to constant folding, algebraic simplifications, and reassociation)