CSE244 Compilers
(a.k.a. Programming Language Translation)

Notes credit go to
me CSE
Laurent Michel CSE
Steven Demurjian CSE
Robert La Barre UTRC

Overview
• Objectives
• Structure of the course
• Evaluation
• Compiler Introduction
  – A compiler bird’s eye view
    • Lexical analysis
    • Parsing
    • Semantic analysis
    • Code generation
    • Optimization
Objectives

- Compilers are….
  - Ubiquitous in Computer Science
  - Central to symbolic processing
  - Relates to
    - Theory of computing
    - Practice of computing
    - Programming languages
    - Operating Systems
    - Computer Architecture

Purpose

- Simple intent
  - Translate
    - From Source Language
    - To Target Language
Translate... Why?

- Languages offer
  - Abstractions
  - At different levels
    - From low
      - Good for machines....
    - To high
      - Good for humans....

Let the computer Do the heavy lifting.

Translate... How?

- Three approaches
  - Interpreted
  - Compiled
  - Mixed
Interpreter

- **Motivation.**
  - Security / Portability.
- **Upside?**
- **Downside?**
- **Phases**
  - Lexical analysis
  - Parsing
  - Semantic checking
  - Interpretation

Compiler

- **Motivation**
  - Speed.
- **Upside?**
- **Downside?**
- **Phases**
  - Code Generation
  - Code Optimization
  - Link & load
Mixed

- **Motivation**
  - The best of two breeds…
- **Upside?**
- **Downside?**

Objectives

- **Learn about compilers because...**
  - It helps to better program
  - Understand the tools of the trade
  - Many languages are compiled / interpreted
    - Programming Languages: [C, Java, ML, LISP,…]
    - Communication Languages: [XML, HTML,…]
    - Presentation Languages: [CSS, SGML,…]
    - Hardware Languages: [VHDL,…]
    - Formatting Languages: [Postscript, troff, LaTeX,…]
    - Query Languages: [SQL & friends]
  - Many tools use compiler technology…
    - Syntax highlighting /
    - Type assist - type completion
  - You may write compiler technology!
Overview

• Objectives
• Structure of the course
• Evaluation
• Compiler Introduction
  – A compiler bird’s eye view
    • Lexical analysis
    • Parsing
    • Semantic analysis
    • Code generation
    • Optimization

Course structure

• A reflection of the topic
  – Lexical analysis
  – Parsing
  – Semantic analysis
  – Midterm
  – Runtime structures
  – Intermediate code generation
  – Machine code generation
  – Optimization
  – Final
Contact Information

• Instructor: Aggelos Kiayias
• Office: ITE Bldg 243
• Office hours: Tuesday 4-6pm, Wed. 5-7pm.
• http://www.cse.uconn.edu/~akiayias

• Class Web-site:
• http://www.cse.uconn.edu/~akiayias/cse244fa06/

• Class Bulletin Board:
• http://cryptodrm.engr.uconn.edu/cse244fa06bb/

Evaluation

• Course evaluation
  – Six homeworks
  – One midterm
  – One final

• Exams
  – Open book
Six Homeworks

• that will all produce one (big fat) project.
• Purpose
  – First Hand Experience with compiler technology
• Six Homeworks are connected
  – Scanning
  – Parsing
  – Analysis
  – IR Code Generation
  – IR Optimization
  – Machine Code Generation

The Source Language
C--

- The Object Oriented Language For Dummies
  - C-- supports
    - Classes
    - Inheritance
    - Polymorphism
    - 2 basic types
    - Arrays
    - Simple control primitives
      - if-then-else
      - while

C-- Example

class Foo {
  int fact(int n) {
    return 0;
  }
  int fib(int x) {
    return 1;
  }
};
class Main extends Foo {
  Main() {
    int x;
    x = fact(5);
  }
  int fact(int n) {
    if (n==0)
      return 1;
    else return n * fact(n-1);
  }
};
The Target Language

- Something realistic...
- Something challenging...
- Something useful...

It’s True!
We will generate code for a Pentium Class Machine Running Either Windows or Linux

But......

- Won’t that be hard?
- No!
  - Our previous C-- compiler (courtesy L. Michel)
    - 6000 lines of code
    - Written in < 10 days
  - Your C-- compiler
    - Will use some of the existing code...
global main
extern printf
extern malloc
section .data
D_0:  dd block_0,block_1
D_1:  dd block_3,block_1,block_2
section .text
main:
  push 4
  call malloc
  add esp,4
  mov [eax], dword D_1
  push eax
  mov eax, dword [eax]
  mov eax, dword [eax+8]
  call eax
  add esp,4

block_0:
  mov [esp-4], dword ebp
  mov ebp, dword esp
  mov [ebp-8], dword esp
  sub esp,8
  mov eax, dword 0
  mov esp, dword [ebp-8]
  mov esp, dword [ebp+8]
  ret

block_1:
  mov [esp-4], dword ebp
  mov ebp, dword esp
  mov [ebp-8], dword esp
  sub esp,8
  mov eax, dword 0
  mov esp, dword [ebp-8]
  mov esp, dword [ebp+8]
  ret

block_3:
  mov [esp-4], dword ebp
  mov ebp, dword esp
  mov [ebp-8], dword esp
  sub esp,20
  mov eax, dword [ebp+8]
  cmp eax,0
  sete ah
  cmp eax,0
  sete ah
  jmp block_5

block_4:
  mov eax, dword 1
  jmp block_6

block_5:
  mov eax, dword [ebp+8]
  sub eax,1
  mov eax, dword [ebp+8]
  push eax
  mov ebx, dword [ebp+4]
  add esp,8
  pop args
  mov ecx, dword [ebp+8]
  smul ecx,eax
  mov eax, dword ecx
  jmp block_6

block_6:
  mov esp, dword [ebp-8]
  mov esp, dword [ebp-4]
  ret

block_2:
  mov [esp-4], dword ebp
  mov ebp, dword esp
  mov [ebp-8], dword esp
  sub esp,12
  push 5
  mov [ebp-4], dword 0
  mov [ebp-4], dword 0
  ret

This is NASM assembly
generated for the
C--
example shown earlier.

Overview

• Objectives
• Structure of the course
• Evaluation
• Compiler Introduction
  – A compiler bird's eye view
    • Lexical analysis
    • Parsing
    • Semantic analysis
    • Code generation
    • Optimization
Compiler Classes

• Compilers Viewed from Many Perspectives

Single Pass
Multiple Pass
Load & Go

Construction

Debugging
Optimizing

Functional

• However, all utilize same basic tasks to accomplish their actions

Compiler Structure

• Two fundamental sets of issues

Analysis
Text analysis
Syntactic analysis
Structural analysis

Synthesis
Program generation
Program optimization

• Our Focus:
  – Both
Some Good news!

- Tools do exist for
  - Lexical and syntactic analysis
  - Note: it was not always the case.
    - Structure / Syntax directed editors:
      - Force “syntactically” correct code to be entered
    - Pretty Printers:
      - Standardized version for program structure (i.e., indenting)
    - Static Checkers:
      - A “quick” compilation to detect rudimentary errors
    - Interpreters:
      - “real” time execution of code a “line-at-a-time”

Phases of compilation

1, 2, 3 : Analysis
4, 5, 6 : Synthesis
Relocatable

1 Pre-Processor
2 Compiler
3 Assembler
4 Relocatable Machine Code
5 Loader Link/Editor

Library, relocatable object files

Analysis

Language Analysis Phases
1 Lexical Analyzer
2 Syntax Analyzer
3 Semantic Analyzer
4 Intermediate Code Generator
5 Code Optimizer
6 Code Generator

Error Handler
Target Program
Lexical analysis

• Purpose
  – Slice the sequence of symbols into tokens

```javascript
Date x := new Date ( System.today( ) + 30 ) ;
```

Syntax Analysis (parsing)

• Purpose
  – Organize tokens in sentences based on grammar
What is a grammar?

• Grammar is a Set of Rules Which Govern
  – The Interdependencies &
  – Structure Among the Tokens

| statement       | is an          | assignment statement, or while statement, or if statement, or ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment statement</td>
<td>is an</td>
<td>identifier := expression ;</td>
</tr>
</tbody>
</table>
| expression      | is an          | (expression), or expression + expression, or expression * expression, or number, or identifier, or ...

Summary so far…

• Turn a symbol stream into a parse tree
Semantic Analysis

• Purpose
  – Determine *Unique / Unambiguous* Interpretation
  – *Catch errors* related to program meaning

• Examples
  – Natural Language
    • “John Took Picture of Mary Out on the Patio”
  – Programming Language
    • Wrong types
    • Missing declaration
    • Missing methods
    • Ambiguous statements…

Semantic Analysis

• Main task
  – Type checking
  – Many Different Situations
    
    \[
    \text{Real := int + char ;}
    \text{while char <> int do}
    \text{.... Etc.}
    \]

  – Primary tool
    • Symbol table
Summary so far...

- Turned a symbol stream into an *annotated* parse tree

Source Program

Lexical Analysis (Lexing)

Token Stream

Syntax Analysis (Parsing)

Parse Tree

Semantic Analysis (Parsing)

Annotated Parse Tree

Analysis

Source Program

1. Lexical Analyzer

2. Syntax Analyzer

3. Semantic Analyzer

Error Handler

Synthesis Phases

4. Intermediate Code Generator

5. Code Optimizer

6. Code Generator

Target Program
Intermediate Code

• What is intermediate code?
  – A *low level* representation
  – A *simple* representation
  – *Easy* to reason about
  – *Easy* to manipulate
  – *Programming Language* independent
  – *Hardware* independent

IR Code example

• Quadruples / 3-address code
  – \((x, y, op, z)\) to represent \(x := y \ op \ z\)
  – Infinitely many temporaries
  – Implicit call stack management

• Example

```javascript
Date x := new Date ( System.today( ) + 30 ) ;
push System
t0 := call today
t1 := 30
t2 := t0 + t1
push t2
t3 := call DateFactory
x := t3
```
**Code Optimization**

- **Purpose**
  - Improve the intermediate code
    - Get rid of redundant / dead code
    - Get rid of redundant computation
    - Reorganize code
    - Schedule instructions
    - Factor out code
  - Improve the machine code
    - Register allocation
    - Instruction scheduling [for specific hardware]

**Machine Code Generation**

- **Purpose**
  - Generate code for the target architecture
  - Code is relocatable
  - Accounts for platforms specific
    - Registers
      - IA32 (i386): 6 GP registers (int)
      - MIPS: 32 GP registers (int)
    - Calling convention
      - IA32: stack-based
      - MIPS: register based
      - Sparc: register sliding window based
Machine code generation

• Pragmatics
  – Generate assembly
  – Let an assembler produced the binary code.

The Entire Process [Analysis]

position := initial + rate * 60

lexical analyzer
id1 := id2 + id3 * 60

syntax analyzer
id1 := id2 + id3 * 60

semantic analyzer
d1 := id2 + id3 * inttoreal 60

Symbol Table
position ....
initial ....
ratre ....

intermediate code generator

Errors
Assemblers

- Assembly code: names are used for instructions, and names are used for memory addresses.

  MOV a, R1
  ADD #2, R1
  MOV R1, b

- Two-pass Assembly:
  - First Pass: all identifiers are assigned to memory addresses (0-offset) e.g. substitute 0 for a, and 4 for b
  - Second Pass: produce relocatable machine code:

    \[ \begin{align*}
    0001 & 01 00 00000000 * \\
    0011 & 01 10 00000010  \text{ relocation bit} \\
    0010 & 01 00 00000100 *
    \end{align*} \]
Linker & Loader

- Loader
  - Input
    - Relocatable code in a file
  - Output
    - Bring executable into virtual address space. Relocate “shared” libs.

- Linker
  - Input
    - Many relocatable machine code files (object files)
  - Output
    - A single executable file with all the object file glued together.
      - Need to relocate each object file as needed

- Want to learn more?
  - CSE258

Pre-processors

- Purpose
  - Macro processing
  - Performs text replacement (editing)
  - Performs file concatenation (#include)
  - Beautify code

- Example
  - In C, #define does not exist
  - In C, #define is handled by a preprocessor
    
    ```
    #define X 3
    #define Y A*B+C
    #define Z getchar()
    ```
Modern Compilers

- **Motto**
  - Re-targetable compilers

- **Organization**
  - Front-end
    - The analysis
    - The generation of intermediate code
  - Flexible IR-code
  - Back-end
    - The generation of machine code
    - The architecture-specific optimization

Modern Compiler Example

- **Gcc**
  - Front-ends for
    - C,C++,FORTRAN,Objective-C,Java
  - IR
    - Static Single Assignment (SSA)
  - Back-ends for
    - IA32, PPC, Ultra, MIPS, ARM, Itanium, DEC, Alpha, VMS, …
Ahead

• Today’s lecture
  – Reading
    • Chapter 1
  – For the inquisitive mind
    • Chapter 2 is an overview of the entire process. Useful to browse and refer to later on.

• Next Lecture
  – Scanning
  – Reading
    • Chapter 3