ECS 142: Compilers

Course Objectives

After completing this course, you should be able to demonstrate your understanding of modern compilers for imperative languages by

- understanding different phases of a compiler, and
- designing and implementing a compiler for a subset of Java.

This Course ···

- is not a "grand tour" of different compilers, interpreters, translators, pre-processors, etc..
- instead, it takes an in-depth look at (i) the theory behind program transformation, and (ii) design and implementation of compilers.
- An important course as it integrates ideas from
  - Language theory (Reg.Exps, CFGs, Pushdown automaton, Parsing, Attributed grammars, etc.)
  - Programming languages (Abstractions, semantics and their implementations)
  - Computer architecture (Assembly programming)
  - Software engineering (reusability, object-oriented software development)

Organizing Theme: Phases

- Study aspects of compilers in terms of different phases, and how each phase is implemented.
- Both theoretical and pragmatic aspects emphasized.

Administrative Matters

Instructor

- Raju Pandey, 3041 Kemper Hall, 752-3584
- Email: `pandey@cs.ucdavis.edu`
- Office Hrs: Tu/Thu. 1:45 PM - 3:00 PM.

TA

- Saigopal Thota
- Email: `sthota@ucdavis.edu`
- Office hours: To be announced.

Course Details

- Midterm: 5 May (In class)
- Final: 4 June, 10:30-12:30, 117 Olson.

Communication

- SmartSite newsgroups
- Course home page: `http://www.cs.ucdavis.edu/ pandey`

Activities

- Programming Project (50%)
  - Java$^-$ $-$: Implement a small subset of Java.
  - Four parts
  - More on this later
- Homeworks (5%)
- Tests (45%)
  - Midterm (20%)
  - Two hours final (25%)

Policies

- Re-grades within one week.
- NO MAKEUP EXAMS.
- NO CHEATING.

Attendance

- Not required
- However, responsible for all material

Textbooks and references

- Primary text book: Dragon book (Aho, Lam, Sethi, and Ullman)
- Reference books:
  - C++ text books (For instance, Lippman)
  - STL handbooks
  - Java (Programming language specification)
  - Lex and Yacc
  - Assembly programming on MIPS machines

Lecture Notes and Reading Material

- Follow text book mostly
- Some additional material from other books - mostly related with project
- Read text book: hard material
- Transparencies available from Course Web Site: `pdf` versions can be accessed from course home page

Computing Resources

- CSIF workstations (Basement EU II)
- Software tools: g++, lex, yacc, make, STL
- PC/Macs okay but need to be able to run your generated code on MIPS machines directly. (SPIM is not sufficient)

Background

- *Theory*: Regular expressions + Context free grammars
- *Language*: Types, Type-equivalence, Semantics of control constructs, Object-orientation.
  Will need to learn (subset of) Java..
- *Architecture*: Microprocessor architectures + Assembly programming.
- *Software development*: C++, make, gdb, STL, Lex, Yacc.

Course project

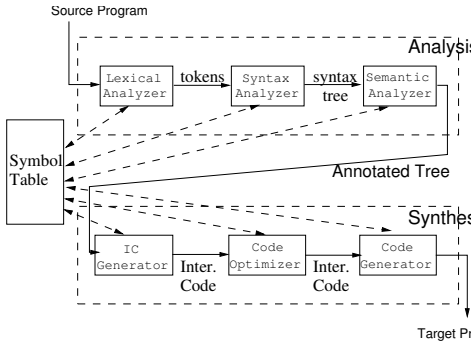
- Problem statement: Write a compiler for a subset of Java.
  Output: MIPS assembly code.
- Substantial software development project.
- Four major parts:
  1. Symbol table (2%).
  2. Syntactic analysis (8%): Use tools (lex and yacc) for writing the lexical analyzer and parser.
  3. Semantic Analysis (25%): Three subparts
     - 3.1 Processing of declarations
     - 3.2 Processing of expressions and statements
     - 3.3 Type checking and other semantic analysis
  4. Code generation (15%): Use MIPS simulator to debug and execute your generated code.
- C++: language for writing the compiler.
- Notes/advise about project:
  - Freely use existing libraries (e.g., STL, LEDA etc.)
  - Develop software at a higher level of abstraction: focus more on putting software components together; less on building from first principle.
  - Be smart and methodical about (i) writing, (ii) maintaining, and (iii) testing your software.
  - Use tools such as make, rcs/sccs, gnu testing tools etc. to help with software development.
  - Use encapsulation, abstraction, inheritance, genericity..

What is a compiler?

- Transform a program represented in a notation (source language) into another program represented in another notation (target language).
  Execute generated program on a machine that can interpret target language
- Focus: Must preserve semantics of original program
- A very general definition of a compiler.
  Note that compiler is independent of application program that it compiles.
- Other examples of tools that transform source program into another source program
  - Pre-processors: transform macros.
  - Translators
- **Interpreters**: Execute the program without explicitly performing the translation.
  Program is like any other data for interpreter.
- **Text formatters** such as TEX, `groff` etc.
- Differences arise in the view of the underlying machine that the translators assume.
  Higher the abstraction, lesser the work a compiler/translator has to do.
- Our focus: Compilers that take a source program, perform analysis on it, and synthesize code that is closer to bare machine (as exemplified by microprocessors).

Distinct Phases of Compilers

- Divided into two major components:
  1. *Analysis* of source program to uncover the structure of the program, and
  2. *Synthesis* of target machine language program that, when executed, will correctly perform the operations of source program.



- Interaction among different phases more complex than shown here.

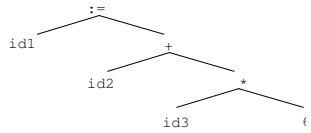
Lexical Analysis

- Break the source program into meaningful units, called tokens.
- Example:
  `position := initial + rate * 60;`
- Compiler sees the above as a sequence of characters, and categorizes them into meaningful units:
  - identifiers: `position, initial, rate`
  - constants: 60
  - operators: `:=, +, *`
  - punctuation: `;`

  Each unit may be assigned a unique identifier (possibly an integer).
- Program is thus now a sequence of tokens.
- Other tasks:
  - Remove extra white spaces
  - Pass over comments
  - Convert all alphabets into one case (in languages that do not distinguish cases)

Syntax Analysis

- Determine the structure of programs, individual expressions, and statements, that is, determine how a program unit is constructed from other units.
- Language: set of legal strings. A program is a string that may and may not belong to the language.
- Language represented by formalism called *context free grammar*. Grammar used to define a set of rules that determine if a sentence belongs to a language.
- A parser uses the grammar to create a representation of a program.



- Parser also detects syntactic errors:
  - Did ';' end the sentence?
  - Do '(' and ')' match?

Semantic Analysis

- Analyze source program for errors that cannot be checked in the earlier phases, mostly semantic errors:
  - Verification of scope rules: is the identifier defined in a scope? If so, where does its definition come from?
    Example: is `position` defined?
  - Type checking: Check if types of operands of operators are valid.
    Example: Can rate be multiplied with $*$?
    Infer types from expressions, operators, and identifier declarations.
- Gather semantic information so that they can be used during code generation phase. Summarize source program information by annotating the tree. May even create a different tree by removing redundant nodes in the parse tree.
- An abstract tree, called syntax tree, may be created whose nodes are operators and leaves are operands.

Symbol Table

- A major component of a compiler. Used to assist various phases.
  Stores information about various symbols as the compiler goes through different phases.
- Various attributes:
  - Storage allocation/address
  - Type
  - Scope
  - If a procedure, its name, number of parameters, etc.

  All attributes that distinguish an identifier.
- Implemented as a data structure containing a record for each identifier. (May have a more complicated structure because of scoping rules but we will see that later.)
- Lexical analyzer phase may create an entry for an identifier in the table.
  Most information added during the semantic analysis phase.

Intermediate Code Generation

- Some compilers generate an explicit intermediate representation of the source program.
  Intermediate representation: Forms an abstract machine.
- Intermediate representation closer to the machine form, and is usually easy to produce. One such representation: "three-address code."
  Three-address code consists of a sequence of instructions, each with at most 3 operands.

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

- Why intermediate form?
  - Portability of compilers: Change the translator from IR to individual machines to port compiler.
  - Write only the front end for different programming languages.
- Project: No intermediate code step.

Code Optimization

- Improve intermediate code:
  - Evaluate constant expressions at compile time: for instance, multiply two constants etc.
  - Delete dead code: code that is never reached.
  - Remove constant expressions from loops
- Example:

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

- Can be very complex and slow as it may involve many passes over the generated code.
  One optimization may lead to another...
- Can involve both local and global optimizations.

Code Generation

- Generation of code that can be directly (or by assembling) executed on the target machine.
- Issues:
  - Selection of memory location for variables
  - Selection of instructions
  - Mapping between registers and variables.
- Compilers may be distinguished according to the kind of machine code that they generate:
  - Pure machine code: Generate code for a specific machine.
  - Augmented machine code: Generate code for a machine architecture augmented with operating system routines and language support routines. Combination of machine+operating system+other routines: *virtual machine*.
  - Virtual machine code: Generated code is composed entirely of virtual instructions. Attractive technique for producing a *transportable* compiler. Also programs can be ported easily.
    Examples: Java byte code representation of Java programs. Also, P-code representation of Pascal programs (first introduced by N. Wirth's group.)

Formats of generated code

- Symbolic Format: Generate code in the assembly format.
  Advantage: cross-compilation. Also check the correctness of the compiler by looking at the output.
  Disadvantage: an extra step as output code must be processed by an assembler after the compiler.
- Relocatable binary format: Binary format in which external references and local instruction and data addresses are not yet bound.
  Addresses assigned related to the beginning of the module or related to symbolically named locations (typical output of assembler.)
  A linkage step is required to add any support libraries and other precompiled routines.
- Memory Image (Load and Go) form: Compiled output loaded in compiler's address space and executed.
  Advantage: Single step to execution. Good for software development where repeated changes are made in the program.
  Disadvantage: Ability to interface with externals, libraries, and pre-compiled routines limited.

Compiler organization alternatives

- Compiler components can be linked together in a variety of different ways.
- Pass: how many times does the compiler need to go over source programs?
- What can make single pass difficult?
  - Space constraints
  - Usage before declaration not required
  - forward gotos, function definitions etc.
- *Single pass for analysis and synthesis:* Scanning, parsing, checking, and translation to target machine code are interleaved.
  - Parser requests tokens from the lexical analyzer as it needs them during the creation of parse tree.
  - As parse tree is constructed, semantic analysis and code generation takes place.
- *One pass analysis and IR synthesis + a code generation pass*
  +ve: flexible (can have multiple code generation phases)
- *Multi-pass analysis:* Used when compilers were required to fit in constrained spaces. Also, where variables/functions are used before they are defined.
- *Multi-pass synthesis:* Allow one or more optimizer passes to transform IR before it is processed by the code generator.