CSE401: Intro to Compiler Construction

Goals
- Learn principles and practice of language translation
- Bring together theory and pragmatics of previous classes
- Understand compile-time vs run-time processing
- Study interactions among
  - Language features
  - Implementation efficiency
  - Compiler complexity
  - Architectural features
- Gain more experience with oo design
- Gain more experience with working in a team
- Gain experience working with SW someone else wrote

Administrivia
- Prerequisites: 322, 326, 341, 378
- Text: *Engineering a Compiler*, Cooper and Torczon, Morgan-Kaufmann 2004
- Course Web is the place to look for materials
- Sign up for mailing list
- Grading:
  - Project 40%
  - Homework 15%
  - MT 15%, Final 25%
  - Class Participation 5% … it’s a cool topic, lock into it

Second Day Homework
- Turn In (On Paper) A Small Profile of Yourself:
  - Photo
  - Email/Year/Major
  - Free time activities
  - An interesting fact about yourself

Project
- Start with a MiniJava compiler in Java … improve it
  - Add:
    - Comments
    - Floating-point values
    - Arrays
    - Static (class) variables
    - For loops
    - Break Statements
    - … And more
  - Completed in stages over the term
  - Strongly encouraged: Work in teams, but only if joint work, not divided work

Compiler Passes

<table>
<thead>
<tr>
<th>Analysis of input program (front-end)</th>
<th>Synthesis of output program (back-end)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character stream</td>
<td>Intermediate Code Generation</td>
</tr>
<tr>
<td>Lexical Analysis</td>
<td>Intermedi ate form</td>
</tr>
<tr>
<td>Token stream</td>
<td>Optimization</td>
</tr>
<tr>
<td>Syntactic Analysis</td>
<td>Intermediate form</td>
</tr>
<tr>
<td>Abstract syntax tree</td>
<td>Code Generation</td>
</tr>
<tr>
<td>Semantic Analysis</td>
<td>target language</td>
</tr>
<tr>
<td>Abstract AST</td>
<td></td>
</tr>
</tbody>
</table>
Example Compilation

Sample (extended) MiniJava program: Factorial.java

```java
// Computes 10! and prints it out
class Factorial {
    public static void main(String[] a) {
        System.out.println(new Fac().ComputeFac(10));
    }
}

class Fac {
    // the recursive helper function
    public int ComputeFac(int num) {
        int numAux;
        if (num < 1) numAux = 1;
        else numAux = num * this.ComputeFac(num-1);
        return numAux;
    }
}
```

First Step: Lexical Analysis

“Scanning”, “tokenizing”
Read in characters, clump into tokens
– strip out whitespace & comments in the process

Specifying tokens: Regular Expressions

Example:

- `Ident ::= Letter AlphaNum*`
- `Integer ::= Digit+`
- `AlphaNum ::= Letter | Digit`
- `Letter ::= 'a' | ... | 'z' | 'A' | ... | 'Z'`
- `Digit ::= '0' | ... | '9'`

Second Step: Syntactic Analysis

“Parsing” -- Read in tokens, turn into a tree based on syntactic structure
– report any errors in syntax

Specifying Syntax: Context-free Grammars

EBNF is a popular notation for CFG’s

Example:

```plaintext
Stmt ::= A Stmt | B Stmt
| ID = Expr Stmt
| ...
Expr ::= Expr + Expr | Expr < Expr | ...
| ! Expr
| Expr . ID ( [Expr {, Expr}] )
| ID
| Integer
| (Expr)
```

EBNF specifies concrete syntax of language; parser constructs tree of abstract syntax of the language

Third Step: Semantic Analysis

“Name resolution and type checking”

- Given AST:
  - figure out what declaration each name refers to
  - perform type checking and other static consistency checks
- Key data structure: symbol table
  - maps names to info about name derived from declaration
  - tree of symbol tables corresponding to nesting of scopes
- Semantic analysis steps:
  1. Process each scope, top down
  2. Process declarations in each scope into symbol table for scope
  3. Process body of each scope in context of symbol table
Fourth Step: Intermediate Code Gen

- Given annotated AST & symbol tables, translate into lower-level intermediate code
- Intermediate code is a separate language
  - Source-language independent
  - Target-machine independent
- Intermediate code is simple and regular
  - Good representation for doing optimizations
  - Might be a reasonable target language itself, e.g. Java bytecode

Example

```java
int Fac.ComputeFac(int this, int num) {
    int t0, t1, t2, t3, t6, t7, t8, t0;
    t0 := 1;
    t1 := num * t0;
    if nonzero t1 goto L0;
    t2 := 1;
    t3 := num - t2;
    t6 := Fac.ComputeFac(this, t3);
    t7 := num * t6;
    numAux := t7;
    goto L2;
    label L0:
    t8 := 1;
    numAux := t8
    label L2:
    return numAux
}
```

Fifth Step: Target Machine Code Gen

Translate intermediate code into target code

- Need to do:
  - Instruction selection: choose target instructions for (subsequences) of IR instructions
  - Register allocation: allocate IR code variables to registers, spilling to memory when necessary
  - Compute layout of each procedures stack frames and other runtime data structures
  - Emit target code