CSE 401: Introduction to Compiler Construction


Goals:

- learn principles & practice of language implementation
- brings together theory & pragmatics of previous courses
- understand compile-time vs. run-time processing
- study interactions among:
  - language features
  - implementation efficiency
  - compiler complexity
  - architectural features
- gain more experience with object-oriented design & Java
- gain more experience working on a team

Prerequisites: 322, 326, 341, 378

Sign up on course mailing list!

Course Outline

Compiler front-ends:

- lexical analysis (scanning): characters → tokens
- syntactic analysis (parsing): tokens → abstract syntax trees
- semantic analysis (typechecking): annotate ASTs

Midterm

Compiler back-ends:

- intermediate code generation: ASTs → intermediate code
- target code generation: intermediate code → target code
  - run-time storage layout
  - target instruction selection
  - register allocation
  - optimizations

Final

Project

Start with compiler for MiniJava, written in Java

Add:

- comments
- floating-point values
- arrays
- static (class) variables
- for loops
- break statements
- and more

Completed in stages over the quarter

**Strongly encourage** working in a 2-person team on project

- but only if joint work, not divided work

Grading based on:

- correctness
- clarity of design & implementation
- quality of test cases

Grading

Project: 40% total
Homework: 20% total
Midterm: 15%
Final: 25%

Homework & projects due at the start of class

3 free late days, per person, for the whole quarter

- thereafter, 25% off per calendar day late
An example compilation

Sample (extended) MiniJava program: Factorial.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:

Stmt ::= if ( Expr ) Stmt | while ( Expr ) Stmt |
       | ID = Expr; |
       | ...

Expr ::= Expr + Expr | Expr < Expr | ...
       | ! Expr
       | Expr . ID ( [Expr , Expr] )
       | ID
       | Integer |
       | (Expr) |
       |

EBNF specifies concrete syntax of language
Parser usually constructs tree representing abstract syntax of language

Third step: semantic analysis

“Name resolution and typechecking”

Given AST:
- figure out what declaration each name refers to
- perform typechecking 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 generation

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

```c
int Fac.ComputeFac(*? this, int num) {
    int T1, numAux, T8, T3, T7, T2, T6, T0;
    T0 := 1;
    T1 := num < T0;
    ifnonzero 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 generation

Translate intermediate code into target code

Need to do:

- instruction selection: choose target instructions for (subsequences of) intermediate code instructions
- register allocation: allocate intermediate code variables to machine registers, spilling excess to stack
- compute layout of each procedure’s stack frame & other run-time data structures
- emit target code

Summary of compiler phases

<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 → Lexical Analysis → Token stream</td>
<td>Intermediate form → Optimization → Intermediate form</td>
</tr>
<tr>
<td>Syntactic Analysis → Abstract syntax tree</td>
<td>Code Generation → Target language</td>
</tr>
<tr>
<td>Semantic Analysis → Annotated AST</td>
<td></td>
</tr>
<tr>
<td>Intermediate Code Generation</td>
<td></td>
</tr>
</tbody>
</table>

Ideal: many front-ends, many back-ends sharing one intermediate language

Other language processing tools

Compilers translate the input language into a different, usually lower-level, target language

Interpreters directly execute the input language

- same front-end structure as a compiler
- then evaluate the annotated AST, or translate to intermediate code and evaluate that

Software engineering tools can resemble compilers

- same front-end structure as a compiler
- then:
  - pretty-print/reformat/colorize
  - analyze to compute relationships like declarations/uses, calls/callees, etc.
  - analyze to find potential bugs
  - aid in refactoring/restructuring/evolving programs

Engineering issues

Compilers are hard to design so that they are

- fast
- highly optimizing
- extensible & evolvable
- correct

Some parts of compilers can be automatically generated from specifications, e.g., scanners, parsers, & target code generators

- generated parts are fast & correct
- specifications are easily evolvable

(Some of my current research is on generating fast, correct optimizations from specifications.)

Need good management of software complexity