Undergraduate Compilers Review and Intro to MJC

Announcements
- Mailing list is in full swing

Today
- Some thoughts on grad school
- Finish parsing
- Semantic analysis
- Visitor pattern for abstract syntax trees

Some Thoughts on Grad School

Goal
- learn how to learn a subject in depth
- learn how to organize a project, execute it, and write about it

Iterate through the following:
- read the background material
- try some examples
- ask lots of questions
- repeat

You will have too much to do!
- learn to prioritize
- it is not possible to read ALL of the background material
- spend 2+ hours of dedicated time EACH day on each class/project
- what grade you get is not the point
- have fun and learn a ton!

Structure of a Typical Compiler

Lexing and Parsing

Lexing
- theoretical tool: regular expressions
- recognizing substrings instead of strings so need longest match and rule priority
- implementation tools: flex, lex, SableCC, etc. generate code that implements a deterministic finite automata that recognizes the specified tokens

Parsing
- theoretical tool: context free grammars
- recognizing a whole program of tokens
- implementation tools: bison, yacc, SableCC, etc. generate a LALR(1) or bottom-up parser that uses shift-reduce parsing to recognize the program and uses syntax-directed translation to generate an AST
Syntactic Analysis (Parsing)

Impose structure on token stream
- Limited to syntactic structure ($\Rightarrow$ high-level)
- Parsers are usually automatically generated from grammars (e.g., yacc, bison, cup, javacc), which use shift-reduce parsing
- An implicit parse tree occurs during parsing as grammar rules are matched
- Output of parsing is usually represented with an abstract syntax tree (AST)

Example

```
for i = 1 to 10 do
  a[i] = x * 5;
```

Syntax-directed Translation: AST Construction example

Grammar with production rules

```
S:  E     {$$ = $1; ;}
E:  E + T {$$ = new node("+", $1, $3); }
   | T        {$$ = $1; }
   | ;
T:  T_ID  {$$ = new leaf("id", $1); }
```

Implicit parse tree for a+b+c

![Implicit parse tree](image)

AST for a+b+c

![AST](image)

Reference: Barbara Ryder’s 198/515 lecture notes

Shift-Reduce Parsing Example (precedence problem)

```
(1) S -> E
(2) E -> E + T
(3) E -> E * T
(4) T -> id
```

Stack | Input | Action
--- | --- | ---
$ | a + b * c | shift
(2) E -> E + T
(3) E -> E * T
(4) E -> T
(5) T -> id

Reference: Barbara Ryder’s 198/515 lecture notes

Bottom-Up Parsing: Shift-Reduce

Grammar

```
S -> E
E -> E + T
E -> E * T
T -> id
```

Rightmost derivation: expand rightmost non-terminals first
SableCC, yacc, and bison generate shift-reduce parsers:
- LALR(1): look-ahead, left-to-right, rightmost derivation in reverse, 1 symbol lookahead
- LALR is a parsing table construction method, smaller tables than canonical LR

Reference: Barbara Ryder’s 198/515 lecture notes
Using SableCC to specify grammar and generate AST

Productions

\[
\text{cststm} ::= \text{cstexp} \\
\text{cstexp} ::= \text{cstexp} \text{tplus} \text{cstterm} \\
\text{cstterm} ::= \text{texp} \\
\text{texp} ::= \text{tid} \\
\]

Abstract Syntax Tree

stm = exp;
exp = (plus) [lExp]; exp [rExp]; exp | (id) tExp;

Example Abstract Syntax Tree MJC

class Factorial {
    public static void main(String[] a) {
        System.out.println(new Factorial().ComputeFac(10));
    }
}

class Factorial {
    public int ComputeFac(int num) {
        int num_aux;
        if (num < 1)
            num_aux = 1;
        else
            num_aux = num *
                (this.ComputeFac(num - 1));
        return num_aux;
    }
}

Semantic Analysis

Determine whether source is meaningful
- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
  - Relate variable uses to their declarations
  - Some semantic analysis takes place during parsing

Example errors (from C)

function1 = 3.14159;
x = 570 + "hello, world!"
scalar[i]
Compiler Data Structures

Symbol Tables
- Compile-time data structure
- Holds names, type information, and scope information for variables

Scopes
- A name space
  - e.g., In Pascal, each procedure creates a new scope
  - e.g., In C, each set of curly braces defines a new scope
- Can create a separate symbol table for each scope

Using Symbol Tables
- For each variable declaration:
  - Check for symbol table entry
  - Add new entry (parsing); add type info (semantic analysis)
- For each variable use:
  - Check symbol table entry (semantic analysis)

Using the Visitor Pattern for semantic analysis

```java
public class DepthFirstAdapter extends AnalysisAdapter {
    ...
    public void inAPlusExp(APlusExp node) {
        defaultIn(node);
    }
    public void outAPlusExp(APlusExp node) {
        defaultOut(node);
    }
    public void caseAPlusExp(APlusExp node) {
        inAPlusExp(node);
        if(node.getLExp() != null) {
            node.getLExp().apply(this);
        }
        if(node.getRExp() != null) {
            node.getRExp().apply(this);
        }
        outAPlusExp(node);
    }
    ...
}
```

Concepts

Compilation stages
- Scanning, parsing, semantic analysis, intermediate code generation, optimization, code generation

Parsing
- generating an AST
- shift-reduce parsing

Semantic Analysis
- symbol tables
- using visitors over the AST

Symbol Table in the MiniJava Compiler

Using Symbol Tables
- For each variable declaration:
  - Check for symbol table entry
  - Add new entry (parsing); add type info (semantic analysis)
- For each variable use:
  - Check symbol table entry (semantic analysis)
Next Time

Reading
- skim Ch 2-6 in Appel
  - focus on 2.1, 2.2, 3.1, 3.3 except parser generation, Ch 4, 5.2, Ch 6
  - skip 3.2 except for FOLLOW description, 3.5, 5.1
- skim Ch 7-9, 12
  - focus on 7.1, 7.3, 8.1, 8.2, 9.3, 12
  - skip 9.2

Lecture
- Finish Undergrad Compilers Review

Parsing Terms (Definitely know these terms)

Lexical Analysis
- longest match and rule priority
- regular expressions
- tokens

CFG (Context-free Grammar)
- production rule
- terminal
- non-terminal
- FOLLOW(X): “the set of terminals that can immediately follow X”

Syntax-directed translation
- inherited attributes
- synthesized attributes

Parsing Terms cont ...

Top-down parsing
- LL(1): left-to-right reading of tokens, leftmost derivation, 1 symbol look-ahead
- Predictive parser: an efficient non-backtracking top-down parser that can handle LL(1)
- More generally recursive descent parsing may involve backtracking

Bottom-up Parsing
- LR(1): left-to-right reading of tokens, rightmost derivation in reverse, 1 symbol lookahead
- Shift-reduce parsers: for example, bison, yacc, and SableCC generated parsers
- Methods for producing an LR parsing table
  - SLR, simple LR
  - Canonical LR, most powerful
  - LALR(1)

BNF (Backus-Naur Form) and EBNF (Extended BNF): equivalent to CFGs