CS536 Lecture 9

Tuesday 17 February 2015

Reminders:

- HW4 up, HW3 due tonight.
- Reading

Last class:

- Fixing Associativity in an ambiguous grammar
- Syntax-Directed Translation

Today:

- Finish Syntax-Directed Translation
- Java CUP

SDT Review

Syntax-Directed Translation

- Consumes a parse tree with actions.
- Actions yield some result.

Abstract Syntax Trees are the result of an SDT for parsing in a compiler.

Implementing ASTs:

How do we represent trees in code?

Building ASTs for Expressions

```
Expr   → Expr + Term      Expr.trans = new PlusNode(Expr2.trans,
                                                    Term.trans)
       | Term             Expr.trans = Term.trans

Term   → Term * Factor    Term.trans = new TimesNode(Term2.trans,
                                                     Factor.trans)
       | Factor           Term.trans = Factor.trans

Factor → INTLIT           Factor.trans = new IntNode(INTLT.value)
       | ( Expr )         Factor.trans = Expr.trans
```

Example: `1 + 2`

ASTs for non-expressions

Example:

```
void foo(int x, int y) {
    if (x == y) {
        return;
    }

    while (x < y) {
        cout << "Hi";
        x = x + 1;
    }
    return;
}
```

Java CUP

Parser Generators: SDT specification to AST

Conceptually similar to JLex for tokens:

- Input: Language rules + Actions
- Output: Java code

Parser spec (`<xxx>.cup`) → Java CUP → Parser source (`parser.java`) + Symbols (`sym.java`)

`Parser.java`:

- Has a constructor that takes in a `Yylex` class
- Has a `parse` method that returns a Symbol whose value field contains the translation of the root nonterminal.
- Uses output from Jlex (scanner and definitions of token value classes)
- Uses definitions of AST classes (`ast.java`)

Java CUP Input Spec

1. Terminal and nonterminal declarations

2. Grammar with rules and actions

3. (Optional) precedence and associativity declarations

Grammar rules:

```
Expr ::= INTLIT
       |  ID
       |  Expr PLUS Expr
       |  Expr TIMES Expr
       |  LPAREN Expr RPAREN
```

Terminals and Nonterminals:

```
terminal     INTLIT;
terminal     ID;
terminal     PLUS;
terminal     TIMES;
terminal     LPAREN;
terminal     RPAREN;
non terminal Expr;
```

Precedence and Associativity:

```
precedence left PLUS;
precedence left TIMES;
precedence nonassoc LESS;
```

Java CUP Example

Assume an `ExpNode` class, with subclasses and members as follows:

- `PlusNode` and `TimesNode`: 2 child `ExpNode`s for operands
- `IDNode`: a `String` field.
- `IntLitNode`: an `int` field.

Also assume token classes `IntLitTokenVal` (with `int` field) for int literal tokens and `IDTokenVal` (with `String` field) for identifier tokens.

Step 1: Add types to terminals (if we need to use a token attribute).

```
terminal IntLitTokenVal INTLIT;
terminal IDTokenVal ID;
terminal PLUS;
terminal TIMES;
terminal LPAREN;
terminal RPAREN;

non terminal ExpNode expr;
```

Note: All nonterminals must have types.

Java CUP Example (cont'd)

Step 2: Add actions to CFG rules.

```
Expr ::= intliteral:i
         {:
             RESULT = new IntLitNode(i.intval);
         :}

       | id:i
         {:


         :}

       | Expr:e1 PLUS Expr:e2
         {:


         :}

       | Expr:e1 TIMES Expr:e2
         {:


         :}

       | LPAREN Expr:e RPAREN
         {:


         :}
     ;
```

Java CUP Example (cont'd)

Java CUP Example 2: Translating Lists

Grammar: `idList → ID | idList COMMA ID`

Given input `x, y, z` the translation should be a `LinkedList` of `String`s:

Parse Tree:

Java CUP Example 2: Translating Lists

CUP spec:

Terminal and nonterminal declarations:

Grammar rules and actions: