CS 536 Announcements for Thursday, February 24, 2022

Programming Assignment 2
• due Friday, February 25

Last Time
• syntax-directed translation
• abstract syntax trees
• implementing ASTs

Today
• Java CUP
• midterm 1 info

Next Time
• wrap up Java CUP
• review

Parser generators

Tools that take an SDT spec and build an AST
• YACC Yet Another C Compiler - creates a parser in C
• Java CUP Constructor of Useful Parsers - creates a parser in Java

Conceptually similar to JLex:
• Input: language rules + actions
• Output: Java code
Java CUP

**parser.java**

- **constructor** takes argument of type `Yylex`
- **parse** method
  - if input correct, returns `Symbol` whose `value` field contains translation of root nonterm
  - if input incorrect, quits on first syntax error
- uses output of JLex
  - depends on scanner and `TokenVal` classes
  - `sym.java` defines the communication language
- uses definitions of AST classes

Parts of Java CUP specification

**Grammar rules with actions:**

```plaintext
expr ::= INTLITERAL
    | ID
    | expr PLUS expr
    | expr TIMES expr
    | LPAREN expr RPAREN
```

**Terminal and nonterminal declarations:**

```plaintext
terminal     INTLITERAL;
terminal     ID;
terminal     PLUS;
terminal     TIMES;
terminal     LPAREN;
terminal     RPAREN;
non terminal expr;
```

**Precedence and associativity declarations:**

```plaintext
precedence left PLUS;
precedence left TIMES;
```
Java CUP Example

Assume:

- Java class `ExpNode` with subclasses `IntLitNode`, `IdNode`, `PlusNode`, `TimesNode`
- `PlusNode` and `TimesNode` each have two children
- `IdNode` has a `String` field (for the identifier)
- `IntLitNode` has an `int` field (for the integer value)
- `INTLITERAL` token is represented by `IntLitTokenVal` class and has field `intVal`
- `ID` token is represented by `IdTokenVal` class and has field `idVal`

Step 1: add types to terminals and nonterminals

```
terminal     INTLITERAL;
terminal     ID;
terminal     PLUS;
terminal     TIMES;
terminal     LPAREN;
terminal     RPAREN;
```

```
non terminal expr; -> non terminal ExpNode expr
```

Add type to terminal if we want to use the value associated with that terminal.

All nonterminals must have types
Java CUP Example (cont.)

Step 2: add actions to CFG rules

```java
expr ::= INTLITERAL {:
    RESULT = new IntLitNode(i.intVal); :
} | ID {:
    RESULT = new IdNode(i.idVal); :
} | expr PLUS expr {:
    RESULT = new PlusNode(e1, e2); :
} | expr TIMES expr {:
    RESULT = new TimesNode(e1, e2); :
} | LPAREN expr RPAREN {:
    RESULT = e ; :
} ;
```

For more:

```java
nonterm ::= rule1
    \[\begin{align*}
    &\text{\#action for this rule} \\
    &\quad \text{RESULT} = \ldots \\
    \end{align*}\]
    \[\begin{align*}
    \text{\#3} \\
    \text{\#3} \\
    \text{\#3}
    \end{align*}\]
```

```
rule2
    \[\begin{align*}
    &\text{RESULT} = \ldots \\
    \text{\#3} \\
    \text{\#3} \\
    \text{\#3}
    \end{align*}\]
```

```java
```
```
Input: $2 + 3$

Java CUP Example (cont.)
Scanning
- general: what does a scanner do; how does it fit into the design of a compiler
- underlying model: FSMs, DFAs vs NFAs, translating NFA \(\rightarrow\) DFA
- specification of a scanner: regular expressions, JLex specifications
  - you do not need to know all of JLex's special characters

Context-Free Grammars
- specification of a language's syntax via a CFG
- derivations (left-most, right-most)
- parse trees
- expression grammars (precedence, associativity)
- list grammars
- ambiguous grammars
- recursive grammar (left recursive, right recursive)

Syntax-Directed Translation
- "plain" translations
  - writing rules of the form "s1.trans ="
  - being able to define translations of any types (integer, AST nodes, etc.)
- Java CUP translations
  - using :xx to name the translation associated with a symbol
  - defining translations by assigning to RESULT

Watch Piazza for info about
- additional homeworks posted on CFGs & SDTs
- sample midterm
- more details about topics