Administration

• Assignment 1
  – Due Next Wednesday January 17\textsuperscript{th} at midnight

• GitHub
  – Read the Piazza post from Felipe
  – Log into the local GitHub
  – Send your info to Felipe
  – Necessary before you can submit assignment 1
Syntax Analysis

• The “job” of syntax analysis is to read the source text and determine its phrase structure.

• Subphases
  – Scanning (Chapter 2)
  – Parsing (Chapter 3)
  – Constructing an AST (Chapter 4)
Syntax Analysis

Dataflow chart

Source Program \rightarrow \text{Scanner} \rightarrow \text{Stream of “Tokens”} \rightarrow \text{Parser} \rightarrow \text{Abstract Syntax Tree}

\text{Stream of Characters} \rightarrow \text{Error Reports}

\text{Error Reports}

Introduction (chapter 1)
1) Scan: Divide Input into Tokens

An example Expression source program:

```
y = 2012;
print y+1
```

Tokens are “words” in the input, for example keywords, operators, identifiers, literals, etc.

<table>
<thead>
<tr>
<th>ident</th>
<th>equals</th>
<th>intlit</th>
<th>semi</th>
<th>print</th>
<th>ident</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>=</td>
<td>2012</td>
<td>;</td>
<td>print</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op.</th>
<th>intlit</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
2) Parse: Determine “phrase structure”

Parser analyzes the phrase structure of the token stream with respect to the grammar of the language.

Program

Print Statement

Statements

Expression

Statement

Expression

Expression

<table>
<thead>
<tr>
<th>Ident</th>
<th>primary-Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ident}</td>
<td>\texttt{y}</td>
</tr>
<tr>
<td>\texttt{eq}</td>
<td>=</td>
</tr>
<tr>
<td>\texttt{intlit}</td>
<td>2012</td>
</tr>
<tr>
<td>\texttt{semi}</td>
<td>;</td>
</tr>
<tr>
<td>\texttt{print}</td>
<td>prin</td>
</tr>
<tr>
<td>\texttt{ident}</td>
<td>y</td>
</tr>
<tr>
<td>\texttt{op}</td>
<td>+</td>
</tr>
<tr>
<td>\texttt{intlit}</td>
<td>1</td>
</tr>
</tbody>
</table>
3) AST: Construct the Abstract Syntax Tree

```plaintext
y = 2012;
print y+1
```

```
Program
  Assign
    IdentifierExp y
    IntegerLiteral 2012
  Print
    Plus
      IdentifierExp y
      IntegerLiteral 1
```
Scanning

- Based on regular expressions
  - 60 years of theory on our side
Regular Expressions

• Regular Expressions are a notation for defining sets of strings

Different kinds of RE:

$\varepsilon$ The empty string
$t$ Generates only the string $t$
$XY$ Generates any string $xy$ such that $x$ is generated by $x$ and $y$ is generated by $Y$
$X \mid Y$ Generates any string which generated either by $X$ or by $Y$
$X^*$ The concatenation of zero or more strings generated by $X$
$(X)$ For grouping
Extended Regular Expressions

- For convenience, we often add additional expressions and operators

**Extended REs:**

- $X?$ Either zero or one string generated by $X$
- $X^+$ The concatenation of one or more strings generated by $X$
- $[a-z]$ Any one character from the specified set
- . Any single character
Regular Expression Examples

• What sets of strings do the following REs generate:

1) ε
2) M r | M s
3) M(r|s)
4) (foo|bar)(foo|bar)*
5) (0|1|2|3|4|5|6|7|8|9)*
6) 0|(1|..|9)(0|1|..|9)*
What is the language of this RE?

\[(0|3|6|9| (1|4|7) (0|3|6|9) * (2|5|8) | (2|5|8| (1|4|7) (0|3|6|9) * (1|4|7)) (0|3|6|9| (2|5|8) (0|3|6|9) * (1|4|7) ) * (1|4|7| (2|5|8) (0|3|6|9 ) * (2|5|8)) ) ) * \]
RE’s for Scanning

• What “kinds” of tokens do “normal” programming languages need?
  – Keywords
  – Literals
  – Punctuation
  – Identifiers
  – Comments
  – White-space
What about conflicts?

• In Java, is “new” a keyword or an identifier?
  – What is the fix?

• In Java, is “abc123” the identifier “abc123” or the identifier “abc” followed by the integer literal 123?
  – What is the fix?
Scanner Generators

• It is possible (in fact, quite straightforward) to generate a lexical analyzer (scanner) automatically from a description based on regular expressions.

• We will use such a tool (JavaCC).
Let’s look at the Expressions scanner JavaCC file

- Package:  
  - parser.jcc

- File:  
  - ExpressionsParser.jj
Parsing Overview

• Recognize phrase structure, or sentential structure
• Different parsing strategies: Top Down vs Bottom Up
• Top Down “predictive” parsers (LL parsing)
  – Manual construction
  – Using JavaCC
• Bottom Up Parsers (LR parsing)
  – Never constructed manually
Parsing

• Based on context free grammars
  – 60 years of theory on our side
Context Free Grammars

• A more powerful abstraction than regular expressions
  – Any language expressible by a RE can be expressed by a CFG but not the other way around!

• Generally: a language that exhibits “self embedding” cannot be expressed by regular expressions

• Programming languages all exhibit “self embedding”, i.e., expressions contain other expressions as sub-components

• Another way to view this is: regular expressions can’t count, but CFGs can
  – $a^n b^n$, or matching ()
BNF: Bachus-Naur Form

• We write context free languages using production rules of the form:
  – LHS ::= RHS
  – Where LHS is a non-terminal
  – RHS is a possibly empty sequence of terminals and non-terminals

• And immediately we extend BNF for convenience
  – Add |, *, ?, +, (), ...
The Expression language CFG

Program ::= Statement* print Expr EOF
Statement ::= Ident = Expr ;
Expr ::= CEexpr ( ? Expr : Expr )?
CEexpr ::= AddExpr ( < AddExpr )?
AddExpr ::= MultExpr ( ( + | − ) MultExpr )*
MultExpr ::= NotExpr ( * NotExpr )*
NotExpr ::= !NotExpr | PrimaryExpr
PrimaryExpr ::= Ident | Literal | ( Expr )
Recursive Descent Parsing

• Also known as “Predictive Parsing”, or more formally, LL parsing (usually, LL(1))

• The parser is able to “predict” what it is looking for after looking at a very short prefix of it (usually 1 token)

• Only some CFGs have the right structure to be parsed this way
JavaCC

• Accepts an EBNF description of a language with embedded Java code, and generates a top-down (or predictive, or LL) parser for it

• The EBNF syntax is somewhat modified from the standard:
  – LHSs have return types and arguments
  – ::= is written as :
  – RHSs are delimited by { and }
  – Java code is included in blocks delimited by { and }
  – And the multiple uses of { and } are rather confusing
Let’s look at the Expressions parser JavaCC file

- **Package:**
  - parser.jcc
- **File:**
  - ExpressionsParser.jj
Building the Parse Tree

• Pretty straightforward tree construction task
• The parser discovers structure from the leaves up
  – At least, if you think about the order in which the functions return
  – So, build the AST incrementally on the return path
• Create a class hierarchy
  – Package:
    • ast
  – Files:
    • AST.java
    • And the rest
AST construction in JavaCC

• Package:
  – parser.jcc

• File:
  – ExpressionsParser.jj
Practical Scanning and Parsing

- What do I really need to know about scanning and parsing to make it work?
  - Scanning?
    - Nothing
    - It “Just Works”
  - Parsing?
    - A few things
    - Limitations of the LL parsing technique
Not all grammars are LL(1)

• There are two common reasons why Top-down Recursive-descent, Predictive parsing won’t work for a particular grammar
  – Left recursion
  – Common prefixes
Left recursion

- If the first symbol on the RHS is the same as the symbol on the LHS this is called left recursion and is bad (for top down parsing)
  - \[ E ::= T \mid E + T \]
- Why is it bad?
- How to fix it?
- Show tongue loading cup/lr-exp.cup
Common prefixes

• If two RHS for the same LHS have a common prefix, this is bad (for top-down parsing)
  – S ::= if E then S | if E then S else S
  – P ::= identifier | literal | ( E ) | identifier ( E )
• Why is it bad?
• How to fix it?
• Show tongue loading cup/lr-exp-fixed.cup
LR parsing

- Look at tongue loading lr-exp-2.exp