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

Flow chart

Source Program \(\rightarrow\) Stream of Characters

\[\text{Scanner} \rightarrow \text{Error Reports}\]

\[\text{Stream of “Tokens”} \downarrow\]

\[\text{Parser} \rightarrow \text{Error Reports}\]

\[\downarrow\]

Abstract Syntax Tree
An example Expression source program:

```plaintext
y = 2014;
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>2014</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.
3) AST: Construct the Abstract Syntax Tree

```plaintext
y = 2014;
print y+1

Program
  Assign
    IdentifierExp y
    IntegerLiteral 2014
  Print
    Plus
      IdentifierExp y
      IntegerLiteral 1
```
Scanning in 15 minutes

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

- Regular Expressions are a notation for defining sets of strings

\[ \varepsilon \] 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 \( Y \)

\[ X | Y \] Generates any string which generated either by \( X \) or by \( Y \)

\[ X^* \] 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) $\varepsilon$
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?

\[(0|3|6|9| (1|4|7) (0|3|6|9)^* (2|5|8) | (2|5|8| (1|4|7) (0|3|6|9)^* (1|4|7)) (0|3|6|9| (2|5|8) (0|3|6|9)^* (1|4|7)) ^* (1|4|7| (2|5|8) (0|3|6|9)^* (2|5|8) ) ) )^* \]
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).
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 in 15 minutes

• Based on context free grammars
  – 60 years of theory on our side
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!
- Regular expressions cannot exhibit “self embedding.”
- Programming languages all exhibit “self embedding”, i.e., expressions contain other expressions, etc.
- e.g., 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 ::= CExpr ( ? Expr : Expr )?
CExpr ::= AddExpr ( < AddExpr )?
AddExpr ::= MultExpr ( ( + | - ) MultExpr )* 
MultExpr ::= NotExpr ( * NotExpr )* 
NotExpr ::= !NotExpr | PrimaryExpr 
PrimaryExpr ::= Ident | Integer | ( 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 an 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
Building the Parse Tree

• Pretty straightforward tree construction task
• The parser discovers structure from the leaves up
  – 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 plus the rest
Recap: What does this Grammar do?

\[
N ::= R \mid P \ R^*
\]
\[
P ::= [1-9]
\]
\[
R ::= [0-9]
\]
Recap: What does this Grammar do?

\[ N ::= R \mid P M \]
\[ M ::= \varepsilon \mid R M \]
\[ P ::= [1-9] \]
\[ R ::= [0-9] \]
Recap: What does this Grammar do?

\[ N ::= 0 \mid [1-9][0-9]^* \]
Recap: What does this Grammar do?

\[ \mathcal{W} ::= \varepsilon \mid [a-z] \mathcal{W} [a-z] \]
Recap: What does this Grammar do?

\[ W ::= \varepsilon \mid a \ W \ a \mid b \ W \ b \mid \ldots \mid z \ W \ z \]
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)
  \[ E ::= T | E + T \]

• Why is it bad?

• How to fix it?
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)
  
  \[ E ::= T | E + T \]

- Why is it bad?

- How to fix it?
  
  \[ E ::= T | T ( + T ) + \]
  
  \[ E ::= T ( + T ) * \]
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)
  \[- E ::= T \mid X \mid E + T \]

- How to fix it?
• 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)
  \[ E ::= T \mid X \mid E + T \]

• How to fix it?
  \[ E ::= T \mid X \mid (T \mid X)(+T)^+ \]
  \[ E ::= (T \mid X)(+T)^* \]
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)
  - $E ::= T | E \ast T | E + T$

- How to fix it?
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)
  - \( E ::= T \mid E \ast T \mid E + T \)

- How to fix it?
  - \( E ::= T \mid T (\ast T \mid + T)^+ \)
  - \( E ::= T (\ast T \mid + T)^\ast \)
Common prefixes

- If two RHS for the same LHS have a common prefix, this is bad (for top-down parsing)
  - \[ E ::= T \mid T \ast E \mid T + E \]

- Why is it bad?
- How to fix it?
Common prefixes

- If two RHS for the same LHS have a common prefix, this is bad (for top-down parsing)
  
  - $E ::= T | T \ast E | T + E$

- Why is it bad?

- How to fix it?
  
  - $E ::= T ( \varepsilon | \ast E | + E )$
  
  - $E ::= T ( \ast E | + E )?"
Common prefixes

• If two RHS for the same LHS have a common prefix, this is bad (for top-down parsing)
  \[ S ::= \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \]

• Why is it bad?

• How to fix it?
  • \[ S ::= \text{if } E \text{ then } S \ (\varepsilon \mid \text{else } S) \]
  • \[ S ::= \text{if } E \text{ then } S \ (\text{else } S) ? \]
Common prefixes

• If two RHS for the same LHS have a common prefix, this is bad (for top-down parsing)
  
  \[ P ::= \text{<Ident> | <Integer> | ( E ) | F} \]
  
  \[ F ::= \text{<Ident> ( E )} \]

• Why is it bad?

• How to fix it?
  
  \[ P ::= \text{<Ident> (( E ))? | <Integer> | ( E )} \]

• But: This can make grammars harder to read for humans!
What does this Grammar do?

\[ S ::= NP \ VP \ . \]

\[ NP ::= \text{the} \ N \]

\[ VP ::= V \ NP? \]

\[ V ::= \text{sings} \mid \text{eats} \]

\[ N ::= \text{cat} \mid \text{bird} \mid \text{song} \]
Context-sensitive checking

• There are two kinds of context-sensitive checking
  – Use of identifiers
  – Type checking
Use of identifiers (symbols)

• Depending on the rules for your language, identifier checking takes either one pass or two
  – Expression language can be done in one pass
  – C can be done in one pass
  – Java requires two, why?
Symbol tables

• Symbol tables associate names found in programs with the thing named
  – In Expressions you can only name constants
    • All you need is the type of the constant and the location of the value (when it is computed)
  – In Java, you can name more things (Which ones?)

• We have to worry about multiple occurrences of the same name that name different things
  – But not in Expressions! (Why?)
Symbol tables in the book

• Functional vs. imperative style symbol tables
  – This is essentially a religious argument
  – Both have their advantages and disadvantages
• The book worries about efficiency of lookup
• The “right” thing to do is probably:
  – create a separate symbol table for every scope (rather than for every binding)
  – have lookup traverse a tree of symbol tables
Type checking

• Done bottom-up
  – The type of an expression is determined completely by the types of its sub-expressions
    • Some languages break this rule – ignore them!
Type checking II

• Based on a collection of type rules
  – $E_1 + E_2$ is correct and has type $\text{int}$ if $E_1$ and $E_2$ are both correct and both have type $\text{int}$
  – $I = E$ is correct if ...
  – $E_1 \ ? \ E_2 : E_3$ is correct and has type ... if ...
  – some operators are polymorphic
    • work on operands with more than one type and can return results with more than one type
• The challenge is ensuring you haven’t forgotten rules
Type checking - Complications

- Subclasses: class A extends B
  - Interfaces: class C implements D
  - Subranges: char, short, int, long, long long

- Type conversions
  - Casts
  - Widening vs. narrowing

- Implicit type conversions
  - double x = 3;
  - double x; int i; x + i