Programming Languages

- Many high level programming languages of different types and for different purposes
  - Procedural/Imperative (C, C++, Fortran, Pascal)
  - Object Oriented (Java, C++)
  - Functional (Lisp, Scheme, ML)
  - Logic (Prolog)
  - Parallel processing
  - Scripting (Perl, Python)
  - Database (Oracle, Mysql)

Questions
- What is a language?
- How do we describe it?
- How can we execute it in the von Neumann architecture?

Languages

- What is a language?
  - A set of symbols that "makes sense"
- Words in English
  - Symbols come from the alphabet
  - English words are those from an accepted dictionary
- English sentences
  - Symbols are words (with appropriate capitalization) and punctuation
  - English sentences are sequences that obey grammatical rules
    (subject, object, verbs, tenses, conjugation, etc.)
- Java programs
  - Symbols are operators, keywords, identifiers, braces, semi-colons, etc.
  - Java programs are sequences that obey rules of Java specification
- Algebraic expressions . . .
Grammars

- How to specify the rules for a language
  - Not so easy for English – many exceptions, arcane constructions, interpretations, dialects
  - Easier for programming languages – has to be more precise since it is checked by a program (the compiler)
    - But even the rules for Java are not simple
  - A formal grammar exactly specifies the rules that define a language
    - The grammar is concerned with syntax – the correct form for a program
    - The grammar does not specify whether the program will execute correctly or is meaningful (semantics)
  - Grammars are the basis for writing programs (like compilers) that recognize syntactically correct input

Syntax Diagrams

- One way to specify a rule is by a syntax diagram
- Consider the language of identifiers in Java
  - Informally, we can state the rule as a string of letters and or digits that does not begin with a digit (and where _ and $ are considered letters)
  - We can also describe this with a syntax diagram:

```
Letter
   `-
   |  
  Digit
   | 
  `- Letter

Letter
   `-
   |  
  Digit
   | 
  `- Letter
```
Grammar Specification

- Grammars can also be specified with a list of definition rules
  - Use the symbol '|' to indicate 'or' (choice among several rules)
  - Use <item> to indicate something being defined (and can also be used in definitions)

- Grammar for Java identifiers:
  
  ```
  <identifier> = <letter> | <identifier> <letter> | <identifier> <digit>
  <letter> = a | b | ... | z | A | B | ... | Z | _ | $
  <digit> = 0 | 1 | ... | 9
  ```

Recursion in Grammars

- Items in a grammar may be defined in terms of themselves (directly or indirectly)
  - An identifier is a letter followed by an identifier
  - An expression is a sum or product
    - A sum is the operator + between two expressions

- Most programming language grammars use recursion in rules
  - Algorithms to perform recognition often mirror the recursion
Recognizing Java Identifiers

- Return true if string is a valid Java identifier

```java
public boolean isIdentifier(String id) {
    if (id.length() == 1) { // Base case
        if (Character.isJavaIdentifierStart(id.charAt(0)))
            return true;
        else
            return false;
    } else { // Recursive case
        if (Character.isJavaIdentifierPart(id.charAt(id.length()-1)))
            return isIdentifier(id.substring(0, id.length()-1));
        else
            return false;
    }
}
```

Palindrome Grammar

- A palindrome is a string that reads the same forwards and backwards, e.g., "racecar", "noon"
- A grammar for palindromes

```latex
<palindrome> = <char> | <char> <palindrome> <char> | ""
<char> = a | b | ... | z | A | B | ... | Z
```

```java
public boolean isPalindrome(String word) {
    if (word.length() <= 1) { // Base case
        return true;
    } else { // Recursive case
        if (word.charAt(0) == word.charAt(word.length()-1))
            return isPalindrome(word.substring(1, word.length()-1));
        else
            return false;
    }
}
```
Language Translation

- A compiler translates from source language to executable code in phases
- **Lexical analysis** translates source text to tokens
  - Tokens are meaningful groups of characters (like identifiers, operators, etc.)
- **Parsing** translates sequence of tokens to a syntax tree
  - Syntax tree represents the logical structure imposed by the grammar of the language
- **Code generation** translates the syntax tree to an intermediate code representation
  - Generalized assembly language like instructions
- **Optimization** translates intermediate code to actual machine code

Lexical Analysis

- Lexical analysis turns a sequence of characters into a sequence of tokens – the "words" of the language
  - A grammar could be used to define tokens
  - White space is ignored – only used to separate tokens
  - Definition of tokens may mean no white space is needed
  - Comments ignored

- 24 characters of input:
  \[ n = \text{element}[++i] + 100; \]
- After lexical analysis, 10 tokens:

<table>
<thead>
<tr>
<th>Token</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ID</td>
</tr>
<tr>
<td>=</td>
<td>operator</td>
</tr>
<tr>
<td>element</td>
<td>ID</td>
</tr>
<tr>
<td>[</td>
<td>Left bracket</td>
</tr>
<tr>
<td>++</td>
<td>operator</td>
</tr>
<tr>
<td>i</td>
<td>ID</td>
</tr>
<tr>
<td>]</td>
<td>Right bracket</td>
</tr>
<tr>
<td>+</td>
<td>operator</td>
</tr>
<tr>
<td>100</td>
<td>number</td>
</tr>
<tr>
<td>;</td>
<td>delimiter</td>
</tr>
</tbody>
</table>
Parsing

- Lexical analysis produces a pure sequence of tokens which are then parsed according to the grammar
  - Beginning with the starting definition in the grammar, choose rules to replace symbols until only tokens are left
  - We have produced a sequence of tokens that is a sentence in the language of the grammar
- Parsing is the process of figuring out a production that will give a particular sentence
  - A compiler takes a program as input and determines if there is a production in the grammar that yields the program
  - If so, it is a syntactically correct program

A Grammar Production

- A simple expression grammar:
  \[
  \text{<expr>} = ( \text{<expr>} \text{<op>} \text{<expr>} ) \mid \text{<val>}
  \text{<val>} = \text{ID} \mid \text{number}
  \text{<op>} = + \mid *
  \]
- A production for \(( (42 + x) \ast \text{rate} )\)
  \[
  \begin{align*}
  \text{<expr>} & \Rightarrow ( \text{<expr>} \text{<op>} \text{<expr>} ) \quad [ \text{<expr>} = ( \text{<expr>} \text{<op>} \text{<expr>} ) ] \\
  & \Rightarrow ( \text{<expr>} \text{<op>} \text{<val>} ) \quad [ \text{<expr>} = \text{<val>} ] \\
  & \Rightarrow ( \text{<val>} \text{<op>} \text{<val>} ) \quad [ \text{<val>} = \text{<val>} ] \\
  & \Rightarrow ( ( \text{<expr>} \text{<op>} \text{<expr>} ) \ast \text{rate} ) \quad [ \text{<expr>} = ( \text{<expr>} \text{<op>} \text{<expr>} ) ] \\
  & \Rightarrow ( ( \text{<val>} \text{<op>} \text{<val>} ) \ast \text{rate} ) \quad [ \text{<val>} = \text{<val>} ] \\
  & \Rightarrow ( ( \text{<expr>} \text{<op>} \text{<val>} ) \ast \text{rate} ) \quad [ \text{<expr>} = \text{<val>} ] \\
  & \Rightarrow ( ( \text{<val>} \text{<op>} \text{<val>} ) \ast \text{rate} ) \quad [ \text{<val>} = \text{<val>} ] \\
  & \Rightarrow ( (42 + x) \ast \text{rate} ) \quad [ \text{<expr>} = \text{number} ]
  \end{align*}
  \]
A Syntax Tree

- Represent the production as a tree structure
  - Leaves are tokens
  - Branch points are definitions in grammar
  - Top of tree is starting definition
  - Read the leaves from left to right to get the production
- Syntax tree for \( ( (42 + x) \times \text{rate}) \)

![Syntax Tree Diagram]

Parsing Algorithms

- Two basic flavors of parsing
  - Top down parsing builds tree beginning with start definition and expands to the leaves
  - Bottom up parsing matches leaves and constructs the tree toward the top
  - Grammars are classified according to complexity of parsing
- Theory and tools of parsing covered in a Compilers course
  - Parsers for "real" languages typically created with programs that write parsing programs
  - Tools also used to write lexical analyzer programs
- Parsers for some simple grammars can be hand coded
  - Recursive descent parsers have method for each definition
  - May use look ahead to check token and decide on rule
  - If alternative rules cannot be differentiated by look ahead, then a stack is used to save place in token stream for returning to restart after a failed alternative
Recursive Descent Parser

// Grammar rules for <expr>
private void parseExpr() throws SyntaxError {
    String token = nextToken();
    if (token.equals("(")){
        matchToken("(");
        parseExpr();
        parseOp();
        parseExpr();
        matchToken(")");
    } else if (isValue(token)) {
        parseValue();
    } else {
        throw new SyntaxError("Unexpected token");
    }
}

Code Generation

- For simple languages and parsers, code may be generated as it is parsed by the recursive functions
- More common for parsing to build syntax tree data structure
  - Semantic analysis can be done by traversing tree
  - E.g., determine if variables are declared and in scope
  - Code can be generated by traversing tree
  - Tree may be rearranged to produce better code
- Syntax tree can be used for all kinds of program analysis or manipulation
  - Flow analysis
  - Detection of unsafe constructs
  - Even rewriting code to style guidelines
Another Expression Grammar

- Usual infix algebraic notation
  - Does not require parentheses, but may use for grouping
  - Maintains proper precedence and associativity of operators
- Easy to write recursive descent parser
  - One method for each definition in grammar
  - An expr is one or more terms separated by +
  - A term is one or more factors separated by *

\[
\begin{align*}
<\text{expr}> & = <\text{expr}> + <\text{term}> \mid <\text{term}> \\
<\text{term}> & = <\text{term}> * <\text{factor}> \mid <\text{factor}> \\
<\text{factor}> & = ( <\text{expr}> ) \mid <\text{val}> \\
<\text{val}> & = \text{ID} \mid \text{number}
\end{align*}
\]

Syntax Tree for Expressions

- Tree structures are defined recursively
  - A tree is made up of subtrees
  - Branch points (nodes) in the tree may have data themselves as well as referring to subtrees
  - Leaves do not reference subtrees
- Tree of binary arithmetic expressions
  - Each node in tree has left and right subtrees
    - The left and right operands for the expression
  - Leaves of the tree are the simple values in the expression
    - In our grammar, identifiers or numbers
- Recursive descent parser builds tree
  - Each method returns the subtree at that point
Expression Tree ADT

// Base class for items in the Expression tree
abstract class Expr {
    
}

// Binary expression with operator
class BinaryExpr extends Expr {
    private String op;
    private Expr left, right;
    public BinaryExpr(Expr l, String o, Expr r)
    { left = l; op = o; right = r; }
}

// Value expression
class ValueExpr extends Expr {
    private String val;
    public ValueExpr(String v) { val = v; }
}

Building the Syntax Tree

Expr parseExpr() throws SyntaxError {
    Expr expr = parseTerm();
    while (hasMoreTokens()) {
        if (nextToken().equals("+")) {
            matchToken("+");
            expr = new BinaryExpr(expr, nextToken(), parseTerm());
        } else break;
    }
    return expr;
}

Expr parseTerm() throws SyntaxError {
    Expr expr = parseFactor();
    while (hasMoreTokens()) {
        if (token.equals("*")) {
            matchToken("*");
            expr = new BinaryExpr(expr, nextToken(), parseFactor());
        } else break;
    }
    return expr;
}
Building a Syntax Tree (cont.)

```java
Expr parseFactor() throws SyntaxError {
    Expr expr;
    if (nextToken().equals("(")) {
        matchToken("(");
        expr = parseExpr();
        matchToken("")
    } else if (isValue(nextToken())) {
        expr = parseValue();
    } else {
        throw new SyntaxError("Unexpected token");
    }
    return expr;
}

Expr parseValue() throws SyntaxError {
    String token = getToken();
    if (!isValue(token))
        throw new SyntaxError("Unexpected token");
    return new ValueExpr(token);
}
```

Traversing a Tree

- **Pre-order traversal**
  - Visit node first
  - Then visit left subtree
  - Then visit right subtree
- **In-order traversal**
  - Visit left subtree
  - Then visit node
  - Then visit right subtree
- **Post-order traversal**
  - Visit left subtree
  - Then visit right subtree
  - Then visit node
- "Visiting" a node means to process the node's data
- "Visiting" a subtree means to process the subtree
  - Recursively if it is not a leaf
abstract class Expr {
  public abstract String asPrefix();  // Preorder traversal
  public abstract String asInfix();   // Inorder traversal
  public abstract String asPostfix(); // Postorder traversal
}

class BinaryExpr extends Expr {
  public String asPrefix() {
    return op + " " + left.asPrefix() + right.asPrefix();
  }
  public String asInfix() {
    return "(" + left.asInfix() + op + right.asInfix() + ")";
  }
  public String asPostfix() {
    return left.asPostfix() + right.asPostfix() + " " + op;
  }
}

class ValueExpr extends Expr {
  public String asPrefix() {
    return val + " ";
  }
  public String asInfix() {
    return val;
  }
  public String asPostfix() {
    return " " + val;
  }
}