Supplemental Materials:
Grammars, Parsing, and Expressions

CS2: Data Structures and Algorithms
Colorado State University

Original slides by Chris Wilcox

Topics
✦ Formal Grammars
✦ Production Rules
✦ Backus-Naur Form
✦ Prefix, Postfix, and Infix
✦ Tokenizing and Parsing
✦ Expression Trees and Conversion
✦ Expression Evaluation

Formal Grammars
✦ Programming languages are defined using grammars with specific properties.
✦ Grammars define programming languages using a set of symbols and production rules.
✦ Formal grammars simplify the interpretation of programs by compilers and other tools.
✦ The formalism is necessary to avoid the ambiguities associated with natural languages.

Definitions
✦ Grammar: the system and structure of a language.
✦ Syntax: A set of rules for arranging and combining language elements (form):
  − For example, the syntax of an assignment statement is variable = expression;
✦ Semantics: The meaning of the language elements and constructs (function):
  − The semantics of an assignment statement is evaluate the expression and store the result in the variable.
Ambiguity

✦ Natural Language:
  “British left waffles on Falklands.”
Did the British leave waffles behind, or is there waffling by the British political left wing?
  “Brave men run in my family.”
Do the brave men in his family run away, or are there many brave men in his ancestry?

Production Rules (Example)

✦ For example, assume the alphabet consists of \( a \) and \( b \), the start symbol is \( S \):
  
  Production Rules:
  1. \( S \rightarrow aSb \)
  2. \( S \rightarrow ba \)

✦ Valid:
  \( ba, abab, aababb, aababbb, \ldots \) or \( a^n b a^n | n \geq 0 \)

✦ Invalid:
  \( a, b, ab, abb, aba, bab, \ldots \) and everything else!

Backus Naur Form (BNF)

✦ BNF stands for either Backus-Naur Form, after two famous computer scientists.
✦ BNF is a metalanguage used to describe the grammar of a programming language
✦ BNF is formal and precise, in particular it is a notation for context-free grammars.
✦ BNF is essential in compiler construction, there are many dialects, but the differences are minor.
✦ BNF, like other grammars, supports recursive definitions.

Production Rules and Symbols

Production rules use BNF-style conventions:

✦ \( ::= \) means equivalence, allows composition
✦ \( <\text{symbol}> \) means needs further expansion
✦ \( x y \) denotes a sequence of \( x \) followed by \( y \)
✦ \( \{x\} \) denotes zero or more occurrences of \( x \)
✦ \( [x] \) means optional, not required to be present
✦ \( x | y | z \) means one of \( x \) or \( y \) or \( z \)
✦ \( * \) means 0 or more occurrences (EBNF)
✦ \( + \) means 1 or more occurrences (EBNF)
Production Rules
(Java Identifiers)

<identifier> ::= <initial> { <initial> | <digits> )*  
<initial> ::= <letter> | _ | $  
<letter> ::= a | b | c | .. z | A | B | C | .. Z  
<digit> ::= 0 | 1 | 2 | .. 9  

+ Valid:  
myInt0, _myChar1, $myFloat2, $_12345, ...  

+ Invalid:  
123456, 123myIdent, %Hello, my-Integer, ...  

Production Rules (Other Java)

✦ From the Oracle documentation, a few more rules in BNF notation for Java:

BasicForStatement ::=  
for (<ForInit> ; <Expression> ; <ForUpdate>) <Statement>

Assignment ::=  
<LeftHand> <AssignmentOp> <Expression>  
AssignmentOp ::=  
= *= /= %= += -  
- <<= >>= &= ^= |=  

Regular Expressions

✦ Syntax used to define search strings, for example by the Linux ‘grep’ command.  
✦ Many other usages, for example Java String split and many other methods accept them.  
✦ Two ways to interpret, 1) as a pattern matcher, or 2) as a specification of syntax.  
✦ Powerful, but the syntax of regular expressions can be difficult to master!

Regex Cheatsheet (1)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Match zero, one or more of previous</td>
<td>Ah* matches &quot;A&quot;, &quot;Ah&quot;, &quot;Ahhhh&quot;</td>
</tr>
<tr>
<td>?</td>
<td>Matches zero or one of previous</td>
<td>Ah? matches &quot;A&quot; or &quot;Ah&quot;</td>
</tr>
<tr>
<td>+</td>
<td>Match one or more of previous</td>
<td>Ah+ matches &quot;Ah&quot;, &quot;Ahh&quot; not &quot;A&quot;</td>
</tr>
<tr>
<td>\</td>
<td>Used to escape a special character</td>
<td>Hungry? matches &quot;Hungry?&quot;</td>
</tr>
<tr>
<td>.</td>
<td>Wildcard, matches any character</td>
<td>do.* matches &quot;drop&quot;, &quot;door&quot;, &quot;dot&quot;</td>
</tr>
<tr>
<td>[]</td>
<td>Matches a range of characters</td>
<td>[a-zA-Z] matches ASCII a-z or A-Z</td>
</tr>
</tbody>
</table>
### Regex Cheatsheet (2)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Matches previous or next character or group</td>
<td>(Mon)(Tues)day matches &quot;Monday&quot; or &quot;Tuesday&quot;</td>
</tr>
<tr>
<td>()</td>
<td>Matches a specified number of occurrences of previous</td>
<td>[0-9]{3} matches &quot;315&quot; but not &quot;31&quot;</td>
</tr>
<tr>
<td>^</td>
<td>Matches beginning of a string</td>
<td>^http matches strings that begin with http, such as a url</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of a string</td>
<td>ing$ matches &quot;exciting&quot; but not &quot;ingenious&quot;</td>
</tr>
</tbody>
</table>

### Regex Examples (1)

- [0-9a-f]+ matches hexadecimal, e.g. ab, 1234, cdef, a0f6, 66cd, ffff, 456affff.
- [0-9a-zA-Z] matches alphanumeric strings with a mixture of digits and letters
- [0-9][0-9]{2,4} matches social security numbers, e.g. 166-11-4433
- [a-zA-Z0-9]+@[a-zA-Z0-9]+.edu|com matches emails, e.g. whoever@gmail.com

### Regex Examples (2)

- b[aeiou]*t matches bat, bet, but, and also boot, beet, etc.
- [$_A-Za-z][$_A-Za-z0-9]* matches Java identifiers, e.g. x, myInteger0, _ident, a01
- [A-Z][a-z]* matches any capitalized word, i.e. a capital followed by lowercase letters
- .u.u.u. uses the wildcard to match any letter, e.g. cumulus

### Infix Expressions

- **Infix** notation places each operator between two operands for binary operators:
  
  \[ A * x * x + B * x + C \] // quadratic equation

- This is the customary way we write math formulas in programming languages.
- However, we need to specify an order of evaluation in order to get the correct answer.
Evaluation Order

- The evaluation order you may have learned in math class is named PEMDAS:

  parentheses → exponents → multiplication → division → addition → subtraction

- Also need to account for unary, logical and relational operators, pre/post increment, etc.

- Java has a similar but not identical order of evaluation, as shown on the next slide.

Reminder: Java Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>parentheses: {}</td>
</tr>
<tr>
<td>2</td>
<td>unary: ++ --</td>
</tr>
<tr>
<td>3</td>
<td>multiplicative: * / %</td>
</tr>
<tr>
<td>4</td>
<td>additive: + -</td>
</tr>
<tr>
<td>5</td>
<td>relational: &lt; &gt; &lt;= =&gt; instanceof</td>
</tr>
<tr>
<td>6</td>
<td>equality: == !=</td>
</tr>
<tr>
<td>7</td>
<td>bitwise AND: &amp;</td>
</tr>
<tr>
<td>8</td>
<td>bitwise exclusive OR: ^</td>
</tr>
<tr>
<td>9</td>
<td>bitwise inclusive OR:</td>
</tr>
<tr>
<td>10</td>
<td>logical AND: &amp;&amp;</td>
</tr>
<tr>
<td>11</td>
<td>logical OR:</td>
</tr>
<tr>
<td>12</td>
<td>ternary: ? :</td>
</tr>
<tr>
<td>13</td>
<td>assignment: += -= *= /= %= &amp;= ^=</td>
</tr>
</tbody>
</table>

Infix Example

- How a Java infix expression is evaluated, parentheses added to show association.

  \[
  z = (y * (6 / x) + (w * 4 / v)) - 2;
  \]

  \[
  z = ((y * (6 / x)) + ((w * 4) / v)) - 2; // parentheses
  \]

  \[
  z = ((y * (6 / x)) + ((w * 4) / v)) - 2; // multiplication (L-R)
  \]

  \[
  z = (((y * (6 / x)) + ((w * 4) / v)) - 2; // division (L-R)
  \]

  \[
  z = (((y * (6 / x)) + ((w * 4) / v))) - 2; // addition (L-R)
  \]

  \[
  z = (((y * (6 / x)) + ((w * 4) / v))) - 2; // subtraction (L-R)
  \]

  \[
  z = (((y * (6 / x)) + ((w * 4) / v))) - 2; // assignment
  \]

Postfix Expressions

- **Postfix** notation places the operator after two operands for binary operators:

  \[
  A x * x + B x * + C // infix version
  \]

  \[
  A * x x + B x * + C // postfix version
  \]

- Also called reverse polish notation, just like a vintage Hewlett-Packard calculator!

- No need for parentheses, because the evaluation order is unambiguous.
Postfix Example
✦ Evaluating the same expression as postfix, must search left to right for operators:

\[
(y \times (6 \div x) + (w \times 4 \div v)) - 2 \quad // \text{original infix}
\]
\[
y \times (6 \div x) \times w \times 4 \div v / + 2 - \quad // \text{postfix translation}
\]

- (y (6 x /) *) w 4 v / + 2 -
- (y (6 x /) *) w 4 v / + 2 -
- (y (6 x /) *) (w 4 *) v / + 2 -
- (y (6 x /) *) (w 4 *) v / + 2 -
- (((y (6 x /) *) (w 4 *) v /) +) 2 -

Calculator
✦ Buttons you would push on a normal calculator: 12, *, 10, =, +, (, 6, *, 6, ) \(= 156\)
✦ Buttons you would push on my vintage calculator: 12\(\uparrow\), 10, *, 6\(\uparrow\), 6, + \(= 156\)
✦ Note the implicit use of a stack \(\uparrow\), and the fact that no parentheses are needed.

Prefix Expressions
✦ Prefix notation places the operator before two operands for binary operators:

\[
A \times x \times x + B \times x + C \quad // \text{infix version}
\]

\[
++ \quad ** A \times x \times B \times C \quad // \text{prefix version}
\]
✦ Also called polish notation, because first documented by polish mathematician.
✦ No need for parentheses, because the evaluation order is unambiguous.
Formatting

✦ Free-format language: program is a sequence of tokens position of tokens unimportant (C, Java)
✦ Fixed-format language: indentation and position of tokens on page is significant (Python)
✦ Case-sensitive languages (C, C++, Java):
  – myInteger differ from MyInteger and MYINTEGER
✦ Case-insensitive languages (Fortran, Pascal):
  – identifiers and reserved words!

Tokens

✦ Tokens are the building blocks of a programming language:
  – keywords, identifiers, numbers, punctuation
✦ The initial phase of the compiler splits up the character stream into a sequence of tokens.
✦ Tokens themselves are defined by a set of production rules.

Expression Trees

✦ Parsing decomposes source code and builds a representation that represents its structure.
✦ Parsing generally results in a data structure such as a tree:

```
(A * x * x) + (B * x) + C;
```

```
+  
|  +  
|  |  
|  |  
|  |  
|  |  

(A x) (x) (x) (x) (x) (C)
```

Parsing

✦ Think about some of the difficulties with respect to parsing:
  – How do identify reserved word and identifiers?
  – How do you extract special characters?
  – For example, take the following expression:

```
int y = (A * x * x) + (B * x) + C;
```

✦ Straightforward parsing with Scanner yields:

```
[int, y =, (A, *, x, *, x), +, (B, *, x), +, C]
```
Infix, Postfix, Prefix Conversion

<table>
<thead>
<tr>
<th>Infix</th>
<th>Postfix</th>
<th>Prefix</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A * B + C / D</td>
<td>A B C D / +</td>
<td>* A B / C D</td>
<td>multiply A and B, divide C by D, add the results</td>
</tr>
<tr>
<td>A * (B + C) / D</td>
<td>A B C * D /</td>
<td>/ A + B C D</td>
<td>add B and C, multiply by A, divide by D</td>
</tr>
<tr>
<td>A * (B + C / D)</td>
<td>A B C D / *</td>
<td>* A + B C D</td>
<td>divide C by D, add B, multiply by A</td>
</tr>
</tbody>
</table>

Expression Trees

<table>
<thead>
<tr>
<th>Infix</th>
<th>Postfix</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>((A * B) + (C / D))</td>
<td>((A B *) (C D /)) +</td>
<td>(+ (* A B) (C D))</td>
</tr>
<tr>
<td>((A * (B + C)) / D)</td>
<td>((A (B C +) *) D) /</td>
<td>(/ (* A (B C)) D)</td>
</tr>
<tr>
<td>(A * (B + (C / D)))</td>
<td>(A B C D / + *)</td>
<td>(* A (+ B C) D)</td>
</tr>
</tbody>
</table>

Practicum (Future)

1. **First Exercise**: parse an expression into a stream of tokens.
2. **Second Exercise**: turn the token stream into an expression tree.
3. **Third Exercise**: convert infix to postfix, postfix to prefix, prefix to infix, etc.
4. **Fourth Exercise**: evaluate expressions using the expression tree and recursion.

What’s Next?

However, we will need stacks, which we have studied, and trees, which we have not:

✦ **Question**: Does the Java Collection framework have support for binary trees? If not, why not?
✦ **Answer**: No, you have to build your own trees using the same techniques as with your linked list.