Supplemental Materials: Grammars, Parsing, and Expressions

CS2: Data Structures and Algorithms
Colorado State University

Original slides by Chris Wilcox,
Updated by Russ Wakefield and Wim Bohm

Topics

- Grammars
- Production Rules
- Prefix, Postfix, and Infix
- Tokenizing and Parsing
- Expression Trees and Conversion
- Expression Evaluation

Grammars

- Programming languages are defined using grammars with specific properties.
- Grammars define programming languages using a set of symbols and production rules.
- Grammars simplify the interpretation of programs by compilers and other tools.
- Grammars 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, or are there many brave men in his ancestry?

Language and Grammar

- A language is a set of sentences: strings of **terminals** -the words `while`, `,`, `<`, ...
- Grammar defines these, using productions

\[ \text{LHS} ::= \text{RHS} \]

Read this as the LHS is defined by RHS
Language and Grammar

LHS ::= RHS

- RHS is a string of terminals and non-terminals
  - Terminals are the words of the language
  - Non-terminals are concepts in the language
  - Non-terminals include java statements
- A sequence of productions creates a sentence when no non-terminal is left

Production Rules (Example)

- Non-terminals produce strings of terminals. For example, non-terminal S produces certain valid strings of a’s and b’s.
  - S ::= aSb
  - S ::= ba
- Valid:
  - ba, abab, aababb, aaababbb, ... or a^nbab^n | n ≥ 0
- Invalid:
  - a, b, ab, abb, aba, bab, ... and everything else!

Example productions

- S ::= aSb or
- S ::= ba
- S → ba
- S → aSb → abab
- S → aSb → aSbb → aababb
- S → a^nbab^n | n ≥ 0
Production Rules and Symbols

- ::= means equivalence, is defined by
- <symbol>* means needs further expansion
- Concatenation
  - xy denotes x followed by y
- Choice
  - x | y | z means one of x or y or z
- Repitition
  - * means 0 or more occurrences
  - + means 1 or more occurrences
- Block Structure: recursive definition
  - A statement can have statements inside it

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)

<Statement> ::= <Assignment> | <ForStatement> | ...
<ForStatement> ::= 
  for ({<ForInit>; <Expression>; <ForUpdate>})
  <Statement>

<Assignment> ::= 
  <LeftHand> <AssignmentOp> <Expression>
<AssignmentOp> ::= 
  = | *= | /= | %= | += ......
Regular Expressions

- An alternative definition mechanism
  - Simpler because non-recursive
- Syntax used to define 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 a syntax.

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>Match 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;dog&quot;, &quot;does&quot;, &quot;doll&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>{n}</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></td>
<td>[0-9]{2,4} matches &quot;12&quot;, &quot;123&quot;, and &quot;1234&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][3]-[0-9][2]-[0-9][4] matches social security numbers, e.g. 166-11-4433
- [a-zA-Z]*@(a-zA-Z.)*(edu|com) matches emails, e.g. whoever@gmail.com

Regex Examples (2)

- b[aeiou]t matches bat, bet, but, and also boot, beet, beat, 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:

  \[ A \times x \times x + B \times x + C; // \text{quadratic equation} \]

- 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>Parentheses</td>
<td>()</td>
</tr>
<tr>
<td>Unary</td>
<td>++ -- + -</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &gt; &lt;= &gt;= instanceof</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>Bitwise exclusive OR</td>
<td>^</td>
</tr>
<tr>
<td>Bitwise inclusive OR</td>
<td></td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
</tr>
<tr>
<td>Ternary</td>
<td>? :</td>
</tr>
<tr>
<td>Assignment</td>
<td>= += -= *= /= %= &amp;= ^=</td>
</tr>
</tbody>
</table>

Associativity

Operators with same precedence:

```
* / 
and 
+ -
```

are evaluated left to right: 2-3*4 = (2-3)*4
Infix Example

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

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

1. \[ z = (y \cdot (6 / x) + (w \cdot 4 / v)) - 2; // parentheses \]
2. \[ z = (y \cdot (6 / x)) + (w \cdot 4 / v) - 2; // multiplication (L-R) \]
3. \[ z = (y \cdot (6 / x)) + (w \cdot 4) / v - 2; // division (L-R) \]
4. \[ z = ((y \cdot (6 / x)) + (w \cdot 4)) / v - 2; // addition (L-R) \]
5. \[ z = ((y \cdot (6 / x)) + (w \cdot 4)) / v) - 2; // subtraction (L-R) \]
6. \[ z = ((y \cdot (6 / x)) + (w \cdot 4)) / v) - 2; // assignment \]

Postfix Expressions

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

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

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

- 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:

\[ (y \cdot (6 / x)) + (w \cdot 4 / v)) - 2 // original infix \]
\[ y 6 x / \times w 4 v / + 2 - // postfix translation \]

1. \[ (y (6 x / *)) w 4 v / + 2 - \]
2. \[ ((y (6 x / *)) w 4 v / + 2 - \]
3. \[ (y (6 x / *)) (w 4 *) v / + 2 - \]
4. \[ (y (6 x / *)) (w 4 *) v / + 2 - \]
5. \[ ((y (6 x / *)) ((w 4 *) v /) + 2 - \]
6. \[ (((y (6 x / *)) ((w 4 *) v /) + 2 - \]
7. \[ (((y (6 x / *)) ((w 4 *) v /) + 2 - \]
Calculator

(12 * 10) + (6 * 6)

- Buttons you would push on a normal calculator: 12, *, 10, =, +, (, 6, *, 6, ) // = 156
- Buttons you would push on my vintage calculator: 12, 10, *, 6, *, 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:

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

  \[
  ++ **A x x B x C // \text{prefix version}
  \]

- 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 differs 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 regular expressions

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 * B * x * + C + // postfix version
```
Tokenizing

Think about some of the difficulties with respect to tokenizing:

– How do you identify reserved words and identifiers?
– How do you extract special characters?
– For example, take the following expression:

\[
\text{int } y = (A \times x \times x) + (B \times x) + C;
\]

– Straightforward parsing with Scanner yields:

\[
[\text{int}, y, =, (,A, \times, x, \times, x), ,+, (,B, \times, x), ,+, C,];
\]

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 \times B + \frac{C}{D} )</td>
<td>( A \times B + \frac{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 \times (B + C) \div D )</td>
<td>( A \times (B + C) \div D )</td>
<td>(* A + B / C D)</td>
<td>add ( B ) and ( C ), multiply by ( A ), divide ( C ) by ( D ), add by ( D )</td>
</tr>
<tr>
<td>( A \times (B + (C \div D)) )</td>
<td>( A \times (B + (C \div 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>

<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>
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.