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, (, x* < ….
- Grammar defines these, using productions

LHS ::= 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*** **::= a*S*b**
  - ***S*** **::= ba**
- Valid:

  ba, abab, aababb, aaababbb, ... *or* $\mathbf{a^n bab^n}$ | $n \geq 0$)
- Invalid:

  a, b, ab, abb, aba, bab, ... *and* everything else!

Example productions

- ***S*** **::= a*S*b or**
- ***S*** **::= ba**

- S → ba
- S → aSb → abab
- S → aSb → aaSbb → aababb
- S → $\mathbf{a^n bab^n}$ | $n \geq 0$

Production Rules and Symbols

- *::=* means equivalence, is defined by
- <*symbol*> means needs further expansion
- **Concatenation**
  - *x y* denotes *x* followed by *y*
- **Choice**
  - *x* | *y* | *z* means one of *x* or *y* or *z*
- **Repitition**
  - * means 0 or more occurences
  - + means 1 or more occurences
- **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)

Symbol	Meaning	Example
*	Match **zero, one or more** of previous	Ah* matches "A", "Ah", "Ahhhhh"
?	Match **zero or one** of previous	Ah? matches "A" or "Ah"
+	Match **one or more** of previous	Ah+ matches "Ah", "Ahh" not "A"
\	Used to **escape** a special character	Hungry\? matches "Hungry?"
.	Wildcard, matches **any** character	do.* matches "dog", "door", "dot"
[ ]	Matches a **range** of characters	[a-zA-Z] matches ASCII a-z or A-Z [^0-9] matches any except 0-9.

Regex Cheatsheet (2)

Symbol	Meaning	Example
|	Matches previous **or** next character or group	(Mon)|(Tues)day matches "Monday" or "Tuesday"
{ }	Matches a specified **number of occurrences** of previous	[0-9]{3} matches "315" but not "31" [0-9]{2,4} matches "12", "123", and "1234"
^	Matches **beginning** of a string.	^http matches strings that begin with http, such as a url.
$	Matches the **end** of a string.	ing$ matches "exciting" but not "ingenious"

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
- $[0\text{-}9]^{\{3\}}$-$[0\text{-}9]^{\{2\}}$-$[0\text{-}9]^{\{4\}}$ matches social security numbers, e.g. 166-11-4433
- $[a\text{-}z]^{+}$@($[a\text{-}z]^{+}$\.)$^{+}$(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\text{-}Za\text{-}z0\text{-}9]^{*}$ matches Java identifiers, e.g. x, myInteger0, _ident, a01
- [A-Z]$[a\text{-}z]^{*}$ 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

parentheses	( )
unary	++ -- + - ~ !
multiplicative	* / %
additive	+ -
shift	<< >>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

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 * (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; // 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 * (6 / x) + (w * 4 / v)) – 2 // original infix
y 6 x / * w 4 * v / + 2 - // postfix translation

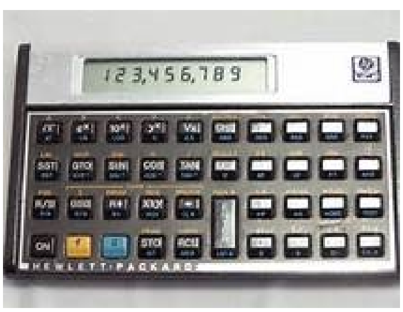
(y (6 x /) *) w 4 * v / + 2 -
((y (6 x /) *) w 4 * v / + 2 -
(y (6 x /) *) (w 4 *) v / + 2 -
(y (6 x /) *) ((w 4 *) v /) + 2 –
((y (6 x /) *) ((w 4 *) v /) +) 2 -
(((y (6 x /) *) ((w 4 *) v /) +) 2 -)

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 (↵), and the fact that no parentheses are needed.

Calculator



Prefix Expressions

- **Prefix** notation places the operator before two operands for binary operators:

A * x * x + B * x + C // infix version

+ + * * A x x * B x C // 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!

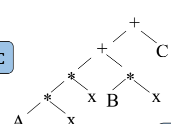
Tokens

- 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 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

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Expression Trees

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

```
        +                    /                 *
       / \                  / \               / \
      /   \                *   D             A   +
     *     /              / \                   / \
    / \   / \            A   +                 B   /
   A   B C   D              / \                   / \
                           B   C                 C   D

 ((A*B)+(C/D))         ((A*(B+C))/D)       (A*(B+(C/D)))
```

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.