Grammars, Regular Expressions and Expression Trees

CS2: Data Structures and Algorithms
Colorado State University

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

Topics

- Grammars
- Production Rules
- Tokenizing
- Regular Expressions
- Expression Trees

Ambiguity in Natural Languages

"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?

Grammars

- Grammars avoid the ambiguities associated with natural languages.
- Programming languages are defined using grammars with specific properties.
- Grammars simplify the interpretation of programs by compilers and other tools.
- Grammars use a set of symbols and production rules.

Definitions

- **Grammar**: System and structure of a language.
- **Syntax**: Set of rules for arranging and combining language elements (*form*):
  - Assignment statement syntax:

```
variable = expression;
```

- **Semantics**: Meaning of the language elements and constructs (*function*):
  - Assignment statement semantics:

    evaluate the expression and store the result in the variable.

Language and Grammar

- A language is a set of sentences:
  - strings of **terminals**
    - the words: *while, (, x <* ....
- Grammar defines these sentences, using production rules

LHS ::= RHS

Read this as the LHS is defined by RHS

Production Rules

LHS ::= RHS

```
<while_statement> ::== 'while' '(' <condition> ')'  '{' <body> '}'
```

- LHS is a non-terminal
- RHS is a string of terminals and non-terminals
  - Terminals are the words of the language
    - For example: while, (, {, },)
  - Non-terminals are concepts in the language
    - For example: condition, body
  - Non-terminals include Java statements
    - For example: while_statement

Generating and Checking Sentences

- A grammar can be used to generate sentences.
  - A sequence of productions creates a sentence when no non-terminal is left
- A grammar can also be used to check whether or not a given sentence is syntactically correct.

Production Rules (Example)

- Non-terminals produce strings of terminals.
  - For example, non-terminal S produces certain valid strings of a's and b's.
    - $\boldsymbol{S}$ **::= a**$\boldsymbol{S}$**b**
    - $\boldsymbol{S}$ **::= ba**
- Valid strings:

  ba, abab, aababb, aaababbb, ... *or* $\mathbf{a^n bab^n}$ | n ≥ 0)
- Invalid strings: cannot be constructed from the rules

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

Example of Using Production Rules

With the following two rules:

- $\boldsymbol{S}$ **::= a**$\boldsymbol{S}$**b or**
- $\boldsymbol{S}$ **::= ba**

We can produce the following strings:

- S → ba
- S → aSb → abab
- S → aSb → aaSbb → aababb
- S → $\mathbf{a^n bab^n}$ | n ≥ 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*
- **Repetition**

  * means 0 or more occurrences

  + means 1 or more occurrences
- **Block Structure**: recursive definition

  A statement can have statements inside it

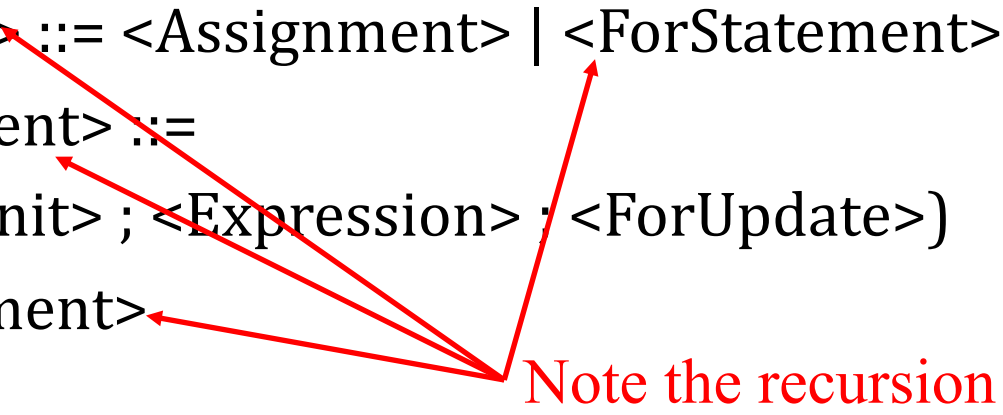
Example Production Rules in Java

<Statement> ::= <Assignment> | <ForStatement> | ...

<ForStatement> ::=

for (<ForInit> ; <Expression> ; <ForUpdate>)

<Statement>



<Assignment> ::=

<LeftHand> <AssignmentOp> <Expression>

<AssignmentOp> ::=

= | *= | /= | %= | += .....

Tokens

- Building blocks of a programming language:
  - keywords, identifiers, numbers, punctuation
- Initial compiler phase splits up the character stream into a sequence of tokens.
- Tokens themselves are defined by regular expressions although one could also use production rules.
- Let's see the production rules first.

Production Rules for 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, ...

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
  - ab, 1234, cdef, a0f6, 66cd, ffff, 456affff.
- [0-9a-zA-Z]+ matches alpha-numeric strings with a mixture of digits and letters
- [0-9]{3}-[0-9]{2}-[0-9]{4} matches social security numbers
  - 166-11-4433
- [a-z0-9]+@([a-z]+\.)+(edu|com) matches emails
  - whoever@gmail.com

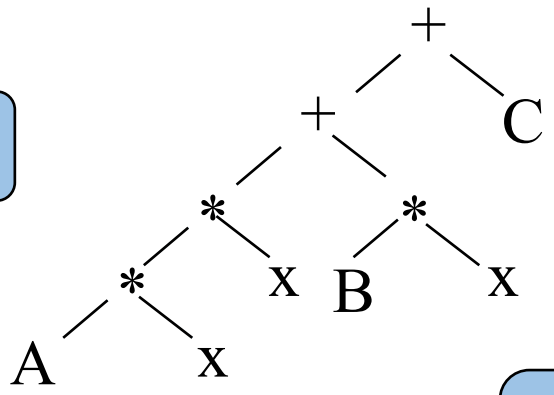
Regex Examples (2)

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

Expression Trees

- Decompose source code and build a representation that represents its structure.
- Results in a data structure such as a tree:

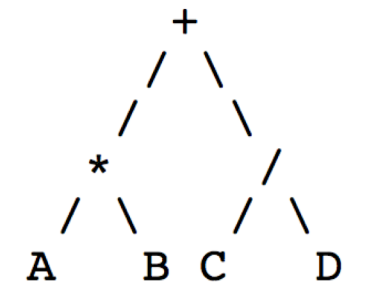
(A * x * x) + (B * x) + C



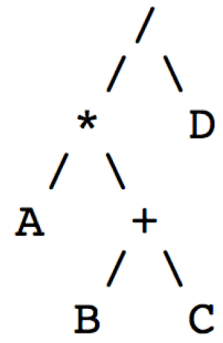
A x * x * B x * + C +
// postfix version

Expression Trees

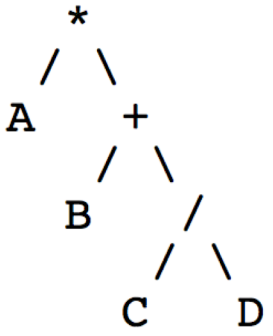
Infix
((A * B) + (C / D))
((A * (B + C)) / D)
(A * (B + (C / D)))



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



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



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

What's next?

- Trees are extremely useful structures in Computer Science.
  - Compilers: Scanning tokens and converting them to expression trees
  - Databases: Efficient indexing for fast retrievals
- Next week we will see how to implement Binary Search Trees