CS 430
Spring 2015

Mike Lam, Professor

Syntax

Overview

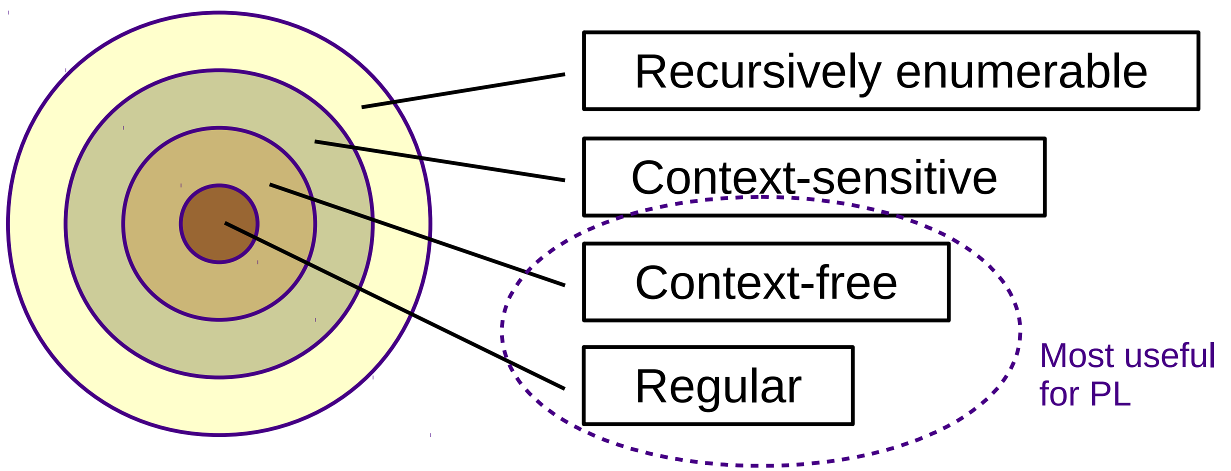
- General topics
  - Syntax (what a program looks like)
  - Semantics (what a program means)
  - Implementation (how a program executes)

Syntax

- Textbook: "the form of [a language's] expressions, statements, and program units."
- In other words:
  - What programs written in that language look like
  - The appearance of the code
- Semantics deal with the meaning of a program
- Syntax and semantics are (ideally) closely related
  - Today we will be studying syntax
- Goals of syntax analysis:
  - Checking for program validity or correctness
  - Facilitate translation (compiler) or execution (interpreter) of a program

Languages

Chomsky Hierarchy of Languages



- Alphabet:
  - Σ = { set of all characters }
- Language:
  - L = { set of sequences of characters from Σ }
  - How to describe L succinctly? Need a *meta-language*.

Lexical Analysis

- *Lexemes* or *tokens*: the smallest building blocks of a language's syntax
- *Lexing* or *scanning*: the process of separating a character stream into tokens

```
total = sum(vals) / n

total      identifier
=          equals_op
sum        identifier
(          left_paren
vals       identifier
)          right_paren
/          divide_op
n          identifier
```

```
char *str = "hi";

char       keyword
*          star_op
str        identifier
=          equals_op
"hi"       str_literal
;          semicolon
```

Lexical Analysis

- Regular expressions
  - Describe regular languages
    - Can be thought of as generalized search patterns
  - Character sets: **[a-z]** or **[0-9]**
  - Concatenation: **ab**
  - Alternation: **a|b**
  - Grouping: **(a|b)c**
  - Quantification: **a*b** (or **a+b** or **a?b**)
    - * = zero or more
    - + = one or more
    - ? = zero or one

Activity

- What languages are described by the following regular expressions:

ab* **a*|b** **a(a|b)*b**

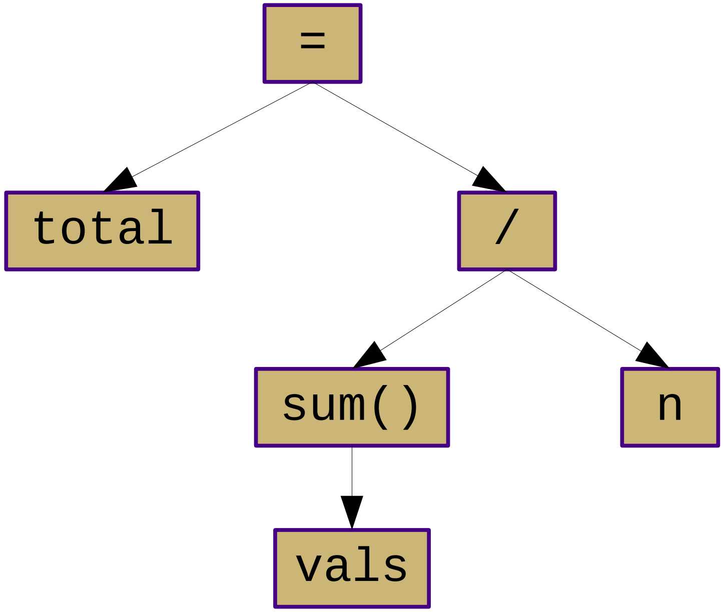
- Write a regular expression that describes decimal numbers
  - Examples: “2.”, “21.3”, “.345”
  - Assume “d” represents a digit ([0-9])

Syntax Analysis

- Tokens have no structure
  - No inherent relationship between each other
  - Need a way to describe hierarchy in a way that is closer to the *semantics* of the language

`total = sum(vals) / n`

```
total      identifier
=          equals_op
sum        identifier
(          left_paren
vals       identifier
)          right_paren
/          divide_op
n          identifier
```



Syntax Analysis

- Context-free language
  - Description of a language's syntax
  - Encodes hierarchy and structure of language tokens
    - Usually represented using a tree
  - Described by *context-free grammars*
    - Usually written in Backus-Naur Form
  - Recognized by *pushdown automata*
    - Brief overview in next lecture
  - Provide ways to control *ambiguity*, *associativity*, and *precedence* in a language

Backus-Naur Form

- *Non-terminals* vs. *terminals*
  - Terminals are essentially tokens
  - One special non-terminal: the *start symbol*
- Production *rules*
  - Left hand side: single non-terminal
  - Right hand side: sequence of terminals and/or non-terminals
  - LHS is replaced by the RHS during generation/derivation
  - Colloquially: "is composed of"
- *Sentence*: a sequence of terminals
  - A sentence is *valid* in a language if it can be derived using the grammar

```
<assign> ::=  <var> = <expr>
<var>    ::=  a | b | c
<expr>   ::=  <expr> + <expr>
           |  <var>
```

```
A →  V = E
V →  a | b | c
E →  E + E
  |  V
```

Derivation

- *Derivation*: a series of grammar-permitted transformations leading to a sentence
  - Each transformation applies exactly one rule
  - Each intermediate string of symbols is a *sentential form*
  - *Leftmost* vs. *rightmost* derivations
    - Which non-terminal do you expand first?
  - *Parse tree* represents a derivation in tree form (the sentence is the sequence of all leaf nodes)
    - Built from the top down during derivation
    - Final parse tree is called *complete* parse tree
    - Represents a program, executed from the bottom up

Example

- Show the leftmost derivation and parse tree of the sentence "a = b + c" using this grammar:

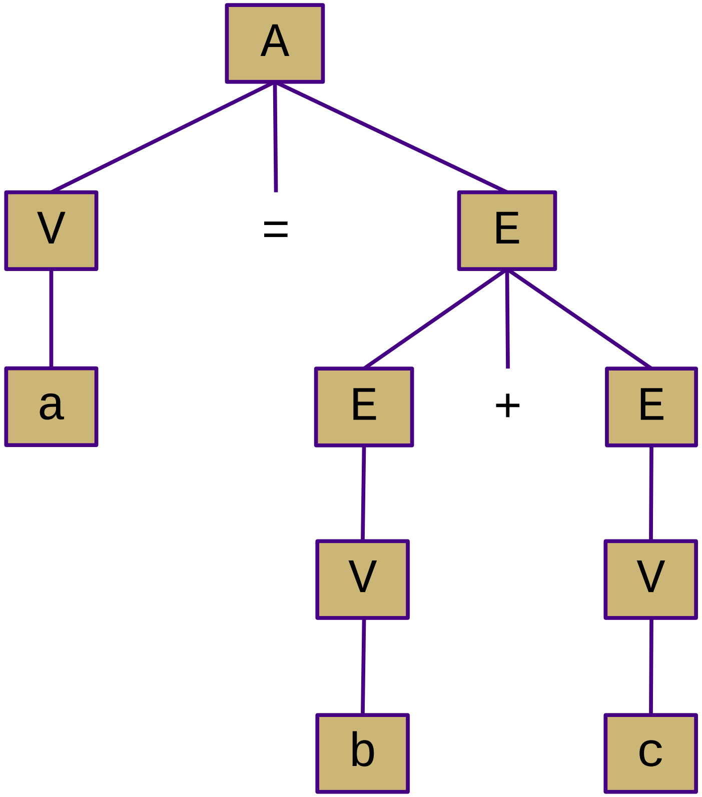
$$
\begin{aligned}
A &\rightarrow V = E \\
V &\rightarrow a \mid b \mid c \\
E &\rightarrow E + E \\
&\mid V
\end{aligned}
$$

Example

- Show the leftmost derivation and parse tree of the sentence "a = b + c" using this grammar:

$$
\begin{aligned}
A &\rightarrow V = E \\
V &\rightarrow a \mid b \mid c \\
E &\rightarrow E + E \\
  &\mid V
\end{aligned}
$$

A
V = E
a = E
a = E + E
a = V + E
a = b + E
a = b + c



Ambiguous Grammars

- An ambiguous grammar allows multiple derivations (and therefore parse trees) for the same sentence
  - The semantics may be similar or identical, but there is a difference syntactically
  - It is important to be precise!
- Can usually be eliminated by rewriting the grammar
  - Usually by making one or more rules more restrictive

Operator Associativity

- The previous ambiguity resulted from an unclear associativity
- Does x+y+z = **(x+y)**+z or x+**(y+z)**?
  - Former is left-associative
  - Latter is right-associative
- Closely related to recursion
  - Left-hand recursion → left associativity
  - Right-hand recursion → right associativity
- Sometimes enforced explicitly in a grammar
  - Sometimes noted with annotations

Operator Precedence

- Precedence determines the relative priority of operators in a single production
- Does x+y*z = **(x+y)***z or x+**(y*z)**?
  - Former: "+" has higher precedence
  - Latter: "*" has higher precedence
- Sometimes enforced explicitly in a grammar
  - Sometimes noted with annotations

Extended BNF

- New constructs
  - Optional: []
  - Closure: {}
  - Multiple-choice: |
- All of these can be expressed using regular BNF
  - (exercise left to the reader)
- So these are really just conveniences

Summary

- Regular languages
  - Described by regular expressions
  - Often used for text processing
  - Core part of languages like Awk, and Perl
- Context-free languages
  - Described by context-free grammars (using BNF)
  - Often used to describe a programming language's syntax
- Lots of very nice language theory
  - We won't dig too deeply in this course
  - Take a compilers or language course if you're interested
    - (or come talk to me)

Activity

1. Draw leftmost and rightmost parse trees for the statement "`x = a + b * c;`" using the following grammar:

```
A → V = E ;
E → E + E
  | E * E
  | V
V → a | b | c ... y | z
```

2. Modify the grammar to make expressions explicitly left-associative.
3. Modify the grammar again to give precedence to operator `*` over `+`.
4. Write a leftmost derivation and a parse tree for the expression "`x = a + b * c;`" using the new grammar.
5. Modify the grammar to allow chained assignments. Is this left- or right-associative?