CS 430 Spring 2019

Mike Lam, Professor

Syntax

Consider the following code

Language A

```
if (a < 5) {
    printf(“%d\n”, a);
}
```

Language B

```
if a < 5:
    print a
```

Language C

```
if [ $a -lt 5 ]; then
    echo $a
fi
```

Language D

```
puts a if a < 5
```

Syntax

- Textbook: syntax is "the form of [a language's] expressions, statements, and program units."
- In other words: the **appearance** of code
- Semantics deal with the **meaning** of code
  - Syntax and semantics are (ideally) closely related
- Goals of syntax analysis:
  - Checking for program validity or correctness
  - Facilitate translation or execution of a program

Languages

- What is a language?
  - More relevantly: what is a formal language?

Searching

- How would you look for all files in the current folder that have the .txt extension?

- How would you look for all files in any subdirectory starting with “cs430”?

These are formal languages!

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*.

- Example:
  - Σ = { a, b, c }
  - L = { “a”, “ab”, “abb”, “abbb”, ... }
  - i.e., “*all strings containing one* ‘a’ *followed by zero or more* ‘b’*s*”

Regular languages

- Regular expressions
  - Describe regular languages
    - Can be thought of as generalized search patterns
  - Quantification: a*
    - * = zero or more
    - + = one or more (extension)
    - ? = zero or one (extension)
  - Concatenation: **ab**
  - Alternation: **a|b**

higher precedence

lower precedence

- Other common features
  - Grouping: **(a|b)c** vs. **a|bc** and (aa)* vs. aa*
  - Character sets: **[a-z]** or **[0-9]** (extension)

Activity

- What languages are described by the following regular expressions?
  - Write down three “words” that are in the language
  - Write down three “words” that are NOT in the language

ab*　　　　**a*|b**　　　　**a(a|b)*b**

Lexical Analysis

- Lexemes or tokens: the smallest building blocks of a language's syntax (described using regular expressions)
- 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
```

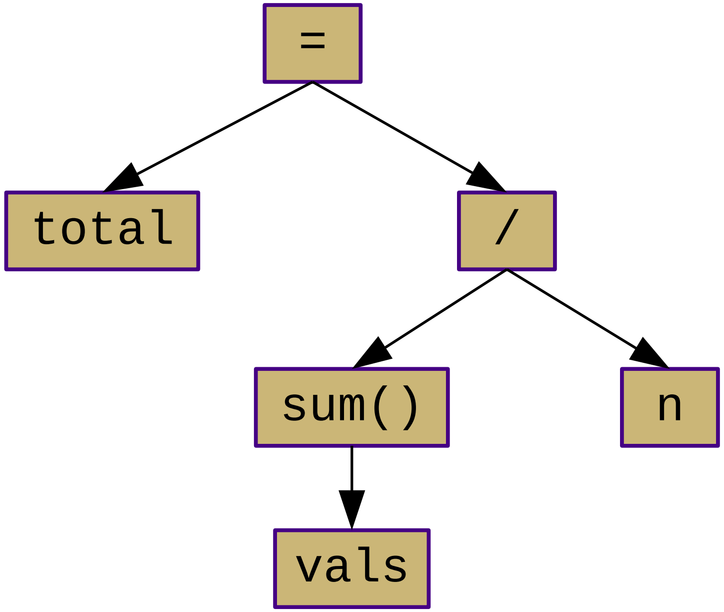
```
char *str = "hi";

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

Syntax Analysis

- Problem: 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`

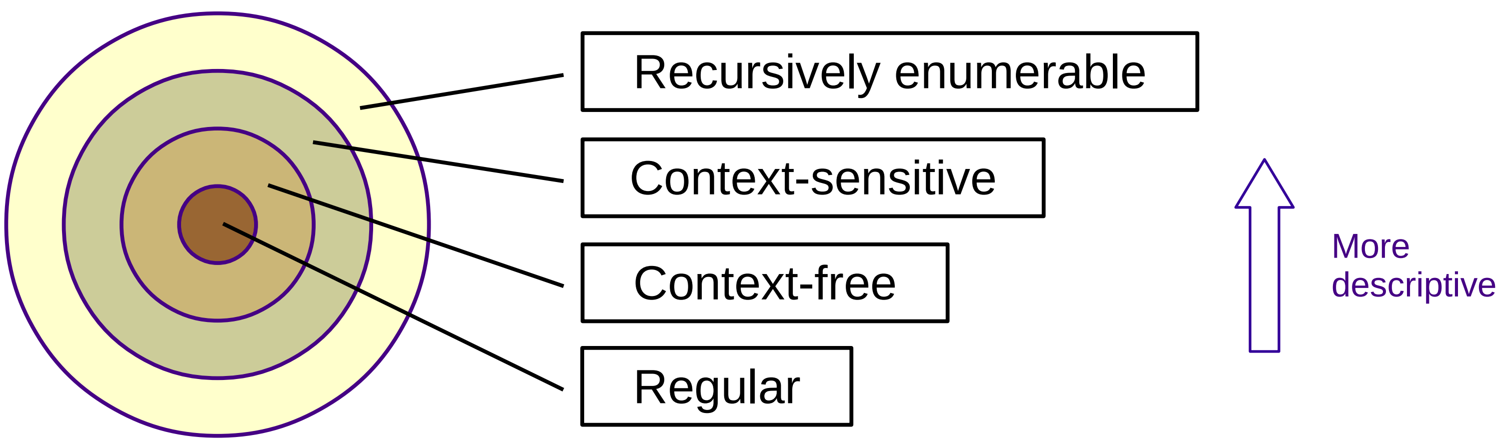
```
total       identifier
=           equals_op
sum         identifier
(           left_paren
vals        identifier
)           right_paren
/           divide_op
n           identifier
```



Syntax Analysis

- Another problem: regular expressions can't "count"
  - Try writing a regular expression that describes strings with matching parenthesis (e.g., "( )" and "(( )) ( )" but not "(( )" or "(( ) ( )))")

Chomsky Hierarchy of Languages



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
  - Provide ways to control ambiguity, associativity, and precedence in a language

Grammars (N,T,P,S)

- Non-terminals (N) vs. terminals (T)
  - Terminals are essentially tokens (described using regular expressions)
  - Non-terminals represent units of program structure
  - One special non-terminal: the start symbol (S)
- Production rules (P): $A \rightarrow (N \cup T)^*$
  - Left hand side: single non-terminal
  - Right hand side: sequence of terminals and/or non-terminals
  - LHS is replaced by the RHS during derivation
  - Colloquially: "is composed of"

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

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

Derivation

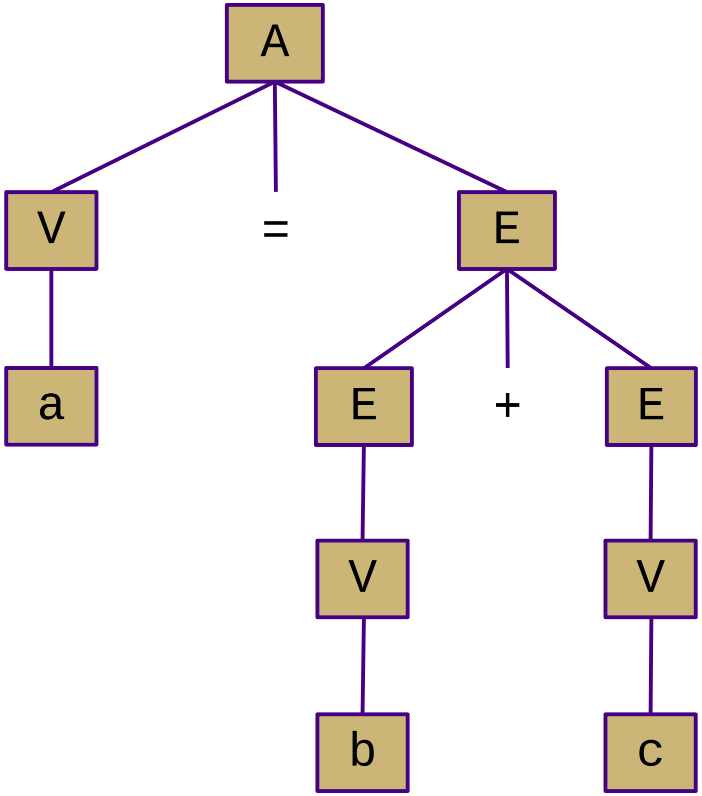
- Derivation: a series of grammar-permitted transformations leading to a sentence (sequence of terminals)
  - Each transformation applies exactly one rule
  - Each intermediate string of symbols is a sentential form
  - Leftmost vs. rightmost derivations
    - Which non-terminal do you expand first?
  - Parse tree represents a derivation in tree form
    - Built from the start symbol (root) down during derivation
    - Final parse tree is called complete parse tree
    - The sentence is the sequence of all leaf nodes (terminals)
    - Interior nodes represent non-terminals
    - 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:

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

A
V = E
a = E
a = E + E
a = V + E
a = b + E
a = b + V
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
- Example: derive "a = x + y + z" and show the parse tree

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

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 ($E \rightarrow E + V$)
  - Latter is right-associative ($E \rightarrow V + E$)
- Can be enforced explicitly in a grammar
  - The problem is the $E \rightarrow E + E$ production
    - Need to remove one possible interpretation
  - Left-associative: change to ($E \rightarrow E + V$)
  - Right-associative: change to ($E \rightarrow V + E$)
  - Sometimes just noted with annotations

Operator Precedence

- Precedence determines the relative priority of operators in a single production
  - Another source of ambiguity
- Does x+y*z = **(x+y)***z or x+**(y*z)**?
  - Former: "+" has higher precedence
  - Latter: "*" has higher precedence
- Can be enforced explicitly in a grammar
  - Separate into two non-terminals (e.g., E and T)
  - Non-terminals closer to the start symbol have lower precedence
    - E.g., for “normal” precedence: $E \rightarrow E + T \mid T$ $T \rightarrow T * V \mid V$
  - Sometimes just 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 CS 432 if you're interested in digging deeper

Examples

- ANTLR grammars:
  - C
  - C++14
  - Java 8
  - Ruby
  - Prolog