CS 430 Spring 2022

Mike Lam, Professor

```
selectionStatement
    :   'if' '(' expression ')' statement ('else' statement)?
    |   'switch' '(' expression ')' statement
    ;

iterationStatement
    :   While '(' expression ')' statement
    |   Do statement While '(' expression ')' ';'
    |   For '(' forCondition ')' statement
    ;
```

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

Case study on importance of making syntax choices carefully:
https://beebo.org/haycorn/2015-04-20_tabs-and-makefiles.html

Syntax Analysis

- Context-free grammar
  - Description of a language's syntax
    - Usually written in Backus-Naur Form
  - Encodes hierarchy and structure of code
    - Usually represented using a tree
  - Provide ways to control ambiguity, associativity, and precedence in a language
  - Four components:
    - Terminals
    - Non-terminals
    - Productions (rules)
    - Start symbol

Grammars

- Non-terminals and terminals
  - Terminals are small chunks of the program code (e.g., “+” or “`foo`”)
  - Non-terminals represent units of program structure
  - 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 derivation
  - Meta-syntax: “→” means "is composed of" and “|” means “or”

More verbose style:

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

More streamlined style:

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

We’ll use this one

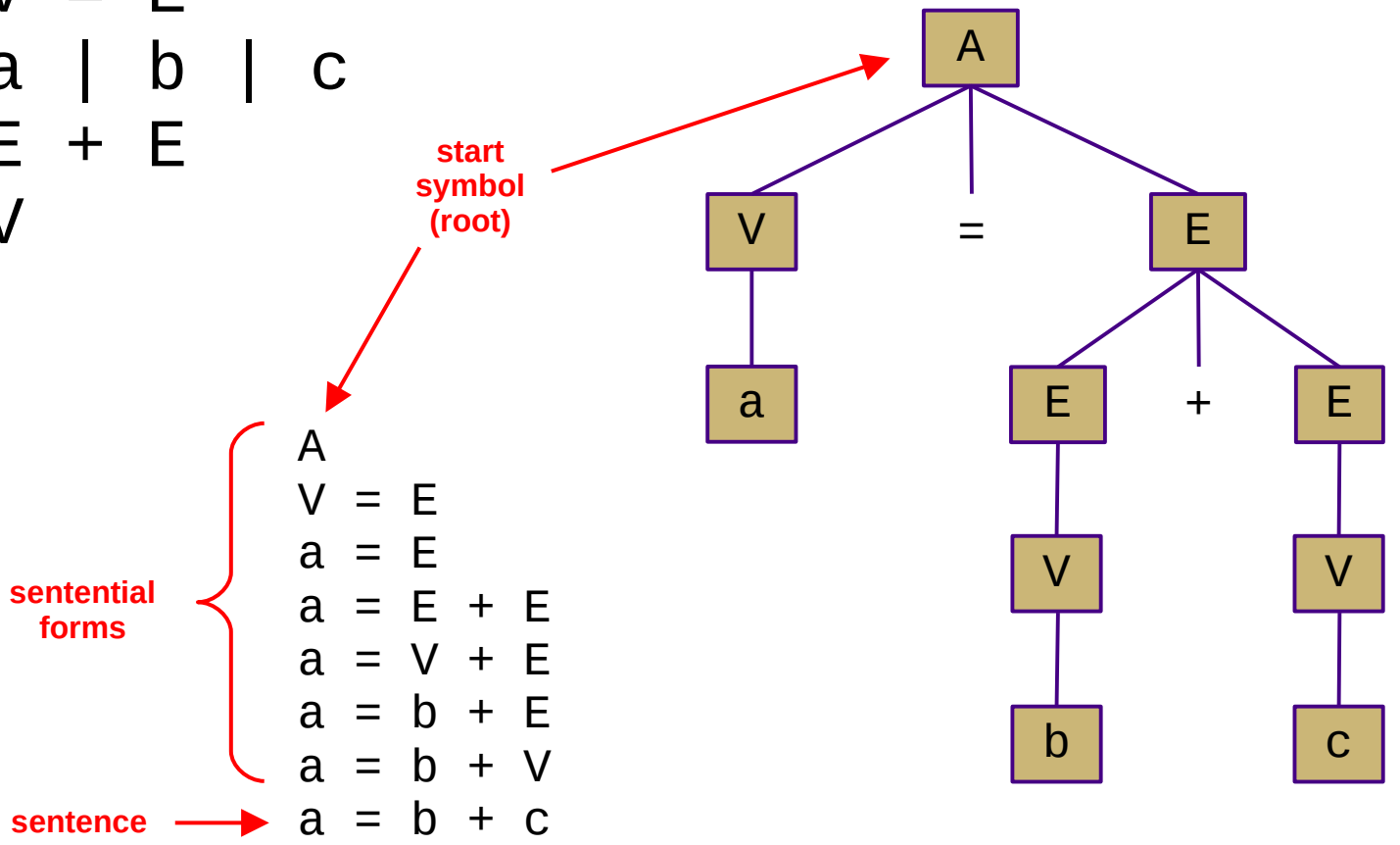
Derivation

- 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:

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



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 “d = a + b + c” and show the parse tree

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

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

Operator Precedence

- Precedence determines the relative priority of operators in a single production (more 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)
    - One non-terminal per level of precedence
    - Non-terminals closer to the root have **lower** precedence
    - E.g., for “normal” precedence: E → E + T | T T → T * V | V
  - Sometimes just noted with annotations
  - Same approach for unary and binary operators
    - For binary operators: left or right associativity?
    - For unary operators: prefix or postfix? (! D vs. D !)
    - For unary operators: is repetition allowed? (C ! vs. D !)

A → A + B
  | B

B → B * C
  | C

C → D !
  | D

Precedence
+ (lowest)
***** (middle)
! (highest)

Extended BNF

- There are many extensions to BNF
  - Most add new meta-syntax operators
- Examples:
  - Optional: []
  - Closure: {} (sometimes w/ superscripts)
  - Multiple-choice: | (already introduced)
- All of these can be expressed using regular BNF
  - (exercise left to the reader)
- So these are really just conveniences

$$E \rightarrow E + E \mid V \quad \equiv \quad E \rightarrow E + E \;\; E \rightarrow V$$

Grammar Examples

```
A → A x
  | x
```

Left Recursive

```
A → x A
  | x
```

Right Recursive

```
A → A + B
  | B
B → C * B
  | C
C → x !
  | x
```

Associativity/Precedence
+ (lowest, binary, left-associative)
* (middle, binary, right-associative)
! (highest, unary, postfix, non-repeatable)

```
A → A + x
  | x
```

Left Associative

```
A → x + A
  | x
```

Right Associative

```
A → A + A
  | A * A
  | x
```

Ambiguous
(Associativity/Precedence)

```
A → B | C
B → x
C → x
```

Ambiguous
(Ad-hoc)

```
A → ifthen A else A
  | ifthen A
  | stmt
```

Ambiguous
("Dangling Else" Problem)

Summary

- Context-free grammars
  - Expressed using Backus-Naur Form
  - Describes a programming language's syntax
  - Controls ambiguity, associativity, and precedence
- Lots of very nice language theory
  - We won't dig too deeply in this course
  - You have seen (or will see) a bit in CS 327
  - Take CS 432 if you're interested in digging deeper

Real-world Examples

- ANTLR grammars:
  - C
  - Java 9
  - Ruby
  - Prolog