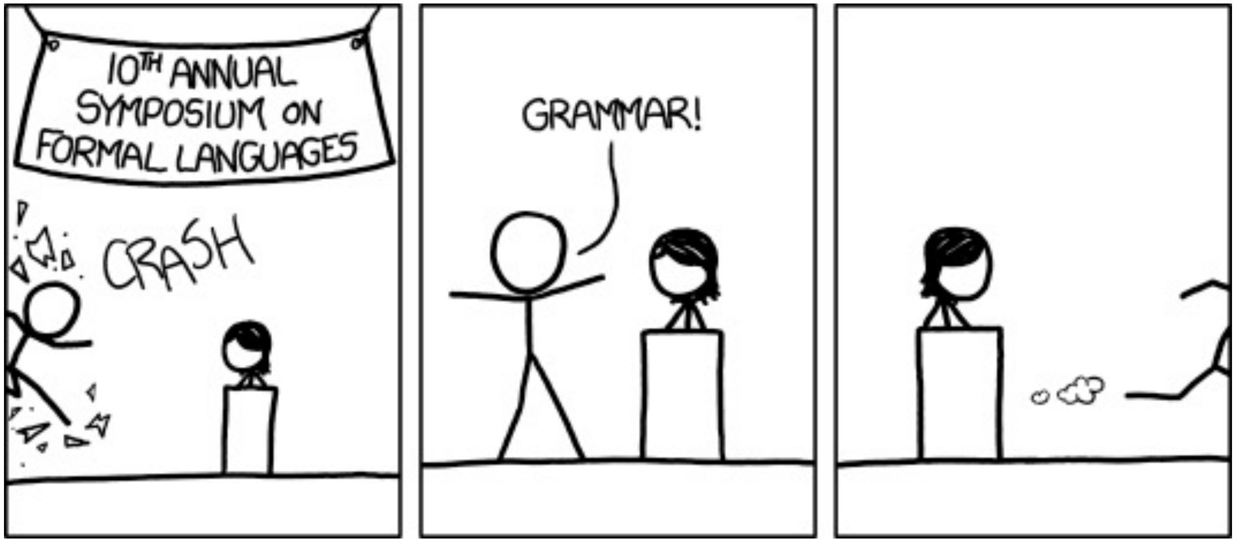
CSE 311: Foundations of Computing

Lecture 21: Context-Free Grammars



[Audience looks around]
“What is going on? There must be some context we’re missing”

Last time: Regular Expressions

Regular expressions over Σ

- **Basis:**

  ε is a regular expression (could also include $\varnothing$)

  a is a regular expression for any $a \in \Sigma$

- **Recursive step:**

  If **A** and **B** are regular expressions then so are:

  $A \cup B$

  AB

  A^*

Last time: Regular Expression is a “pattern”

ε matches the **empty string**

a matches the one character string *a*

A ∪ B matches all strings that either **A** matches or **B** matches (or both)

AB matches all strings that have a first part that **A** matches followed by a second part that **B** matches

A* matches all strings that have any number of strings (even 0) that **A** matches, one after another

Definition of the *language* matched by a regular expression

Limitations of Regular Expressions

- Not all languages can be specified by regular expressions
- Even some easy things like
  - Palindromes
  - Strings with equal number of 0’s and 1’s
- But also more complicated structures in programming languages
  - Matched parentheses
  - Properly formed arithmetic expressions
  - etc.

Context-Free Grammars

- A Context-Free Grammar (CFG) is given by a finite set of substitution rules involving
  - Alphabet Σ of *terminal symbols* that can't be replaced
  - A finite set **V** of *variables* that can be replaced
  - One variable, usually **S**, is called the *start symbol*

- The substitution rules involving a variable **A**, written as

$$\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$$

where each w_i is a string of variables and terminals

– that is $w_i \in (\mathbf{V} \cup \Sigma)^*$

How CFGs generate strings

- Begin with "**S**"

- If there is some variable **A** in the current string, you can replace it by one of the w's in the rules for **A**
  - $\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$
  - Write this as $x\mathbf{A}y \Rightarrow xwy$
  - Repeat until no variables left

- The set of strings the CFG describes are all strings, containing no variables, that can be *generated* in this manner after a finite number of steps

Example Context-Free Grammars

Example: $\mathbf{S} \rightarrow 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

Example Context-Free Grammars

Example: $\mathbf{S} \to 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

0*1*

Example Context-Free Grammars

Example: $\mathbf{S} \to 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

0*1*

Example: $\mathbf{S} \to 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

Example Context-Free Grammars

Example: $\mathbf{S} \rightarrow 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

0*1*

Example: $\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

The set of all binary palindromes

Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(i.e., matching `0*1*` but with same number of 0's and 1's)

Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(i.e., matching 0*1* but with same number of 0's and 1's)

$$\mathbf{S} \rightarrow 0\mathbf{S}1 \mid \varepsilon$$

Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(i.e., matching `0*1*` but with same number of 0's and 1's)

$$\mathbf{S} \rightarrow 0\mathbf{S}1 \mid \varepsilon$$

Grammar for $\{0^n 1^{2n} : n \geq 0\}$

Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(i.e., matching `0*1*` but with same number of 0's and 1's)

$$\mathbf{S} \rightarrow 0\mathbf{S}1 \mid \varepsilon$$

Grammar for $\{0^n 1^{2n} : n \geq 0\}$

$$\mathbf{S} \rightarrow 0\mathbf{S}11 \mid \varepsilon$$

Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(i.e., matching `0*1*` but with same number of 0's and 1's)

S $\rightarrow$ 0**S**1 | ε

Grammar for $\{0^n 1^{n+1} 0 : n \geq 0\}$

Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(i.e., matching `0*1*` but with same number of 0’s and 1’s)

$$\mathbf{S} \rightarrow 0\mathbf{S}1 \mid \varepsilon$$

Grammar for $\{0^n 1^{n+1} 0 : n \geq 0\}$

$$\mathbf{S} \rightarrow \text{A}10$$

$$\mathbf{A} \rightarrow 0\mathbf{A}1 \mid \varepsilon$$

Example Context-Free Grammars

Example: $S \rightarrow (S) \mid SS \mid \varepsilon$

Example Context-Free Grammars

Example: $S \rightarrow (S) \mid SS \mid \varepsilon$

The set of all strings of matched parentheses

Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s

(not just 0^n1^n, also 0101, 0110, etc.)

$$S \to SS \mid 0S1 \mid 1S0 \mid \varepsilon$$

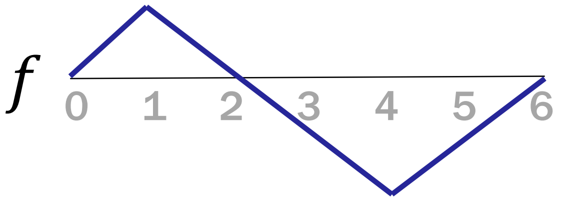
Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s
(not just 0^n1^n, also 0101, 0110, etc.)

$$S \rightarrow SS \mid 0S1 \mid 1S0 \mid \varepsilon$$

Let $x \in \{0,1\}^*$. Define $f_x(k)$ to be #0s – #1s in the first k characters of x.

E.g., for x = 011100



Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s

(not just 0^n1^n, also 0101, 0110, etc.)

$$S \rightarrow SS \mid 0S1 \mid 1S0 \mid \varepsilon$$

Let $x \in \{0,1\}^*$. Define $f_x(k)$ to be #0s – #1s in the first k characters of x.

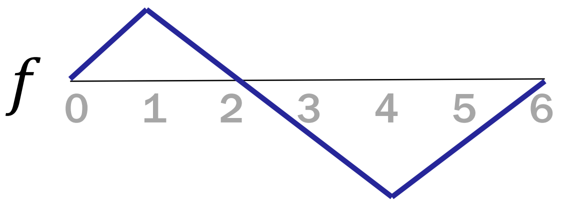
If k-th character is 0, then $f_x(k) = f_x(k-1) + 1$

If k-th character is 1, then $f_x(k) = f_x(k-1) - 1$

Example Context-Free Grammars

Let $x \in (0 \cup 1)^*$. Define $f_x(k)$ to be the number 0s minus the number of 1s in the k characters of x.

E.g., for x = 011100



$f_x(k) = 0$ when first k characters have #0s = #1s

– starts out at 0 $f_x(0) = 0$

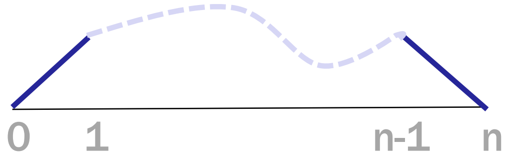
– ends at 0 $f_x(n) = 0$

Example Context-Free Grammars

Three possibilities for $f_x(\mathrm{k})$ for $k \in \{1, \dots, n-1\}$

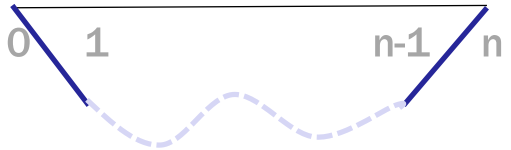
- $f_x(k) > 0$ for all such k

  S → 0S1



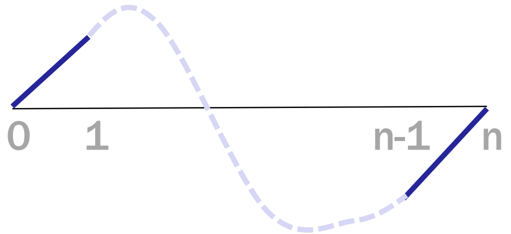
- $f_x(k) < 0$ for all such k

  S → 1S0



- $f_x(k) = 0$ for some such k

  S → SS



Simple Arithmetic Expressions

$E \rightarrow E+E \mid E*E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4$
$\mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Generate $(2*x) + y$

Simple Arithmetic Expressions

$$E \to E+E \mid E*E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Generate $(2*x) + y$

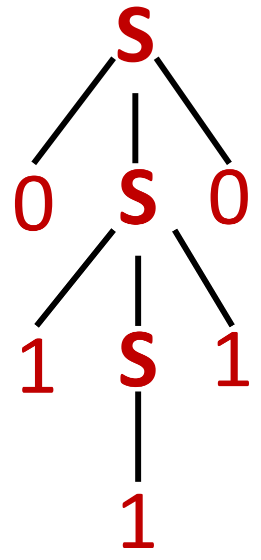
$$E \Rightarrow E+E \Rightarrow (E)+E \Rightarrow (E*E)+E \Rightarrow (2*E)+E \Rightarrow (2*x)+E \Rightarrow (2*x)+y$$

Parse Trees

Suppose that grammar G generates a string x

- A *parse tree* of x for G has
  - Root labeled S (start symbol of G)
  - The children of any node labeled A are labeled by symbols of w left-to-right for some rule A → w
  - The symbols of x label the leaves ordered left-to-right

$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$



Parse tree of 01110

Simple Arithmetic Expressions

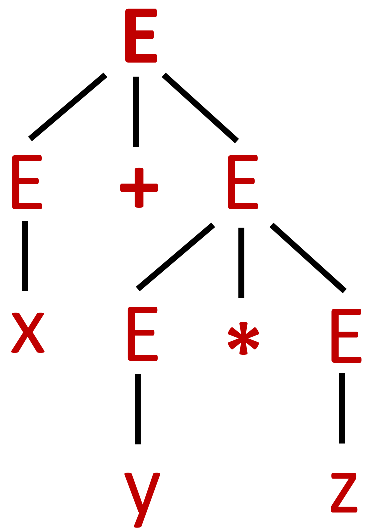
$$E \to E+E \mid E*E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Generate $x+y*z$ in two ways that give two *different* parse trees

Simple Arithmetic Expressions

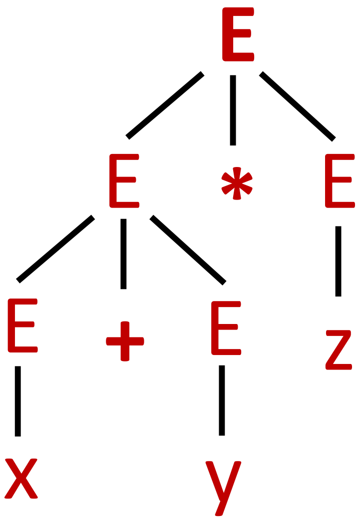
$$E \to E+E \mid E*E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Generate $x+y*z$ in ways that give two *different* parse trees



$E \Rightarrow E+E \Rightarrow x+E \Rightarrow x+E*E \Rightarrow x+y*E \Rightarrow x+y*z$

(multiply y with z and then add to x)



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow x+E*E$

$\Rightarrow x+y*E \Rightarrow x+y*z$

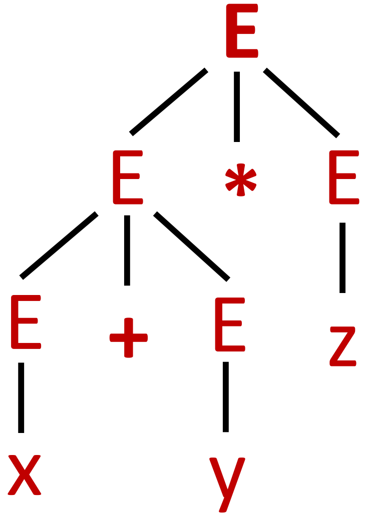
(add x to y, then multiply by z)

building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

E → **T** | **E**+**T**
T → **F** | **F**∗**T**
F → (**E**) | **I** | **N**
I → x | y | z
N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

No longer allows:



building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

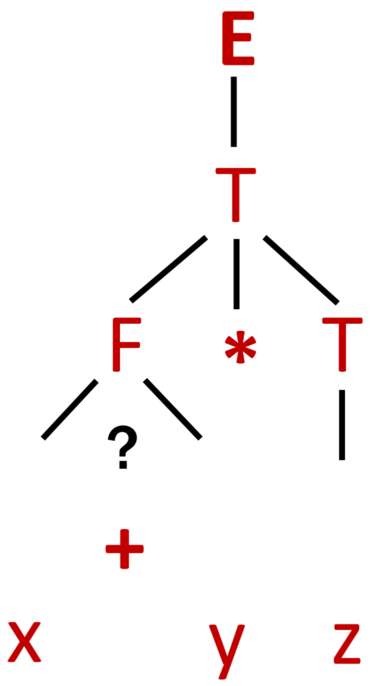
E → **T** | **E**+**T**

T → **F** | **F**∗**T**

F → (**E**) | **I** | **N**

I → x | y | z

N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

E → **T** | **E**+**T**

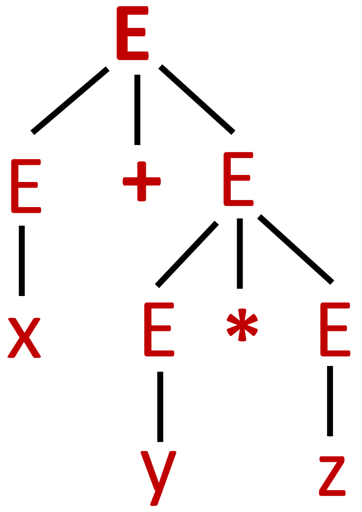
T → **F** | **F**∗**T**

F → (**E**) | **I** | **N**

I → x | y | z

N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Still allows:



building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

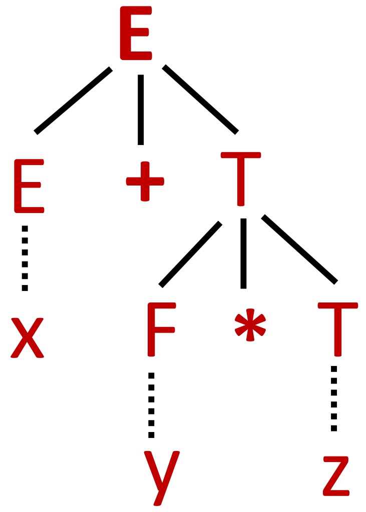
$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E}+\mathbf{T}$

$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F}*\mathbf{T}$

$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$

$\mathbf{I} \rightarrow x \mid y \mid z$

$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



CFGs and recursively-defined sets of strings

- A CFG with the start symbol **S** as its *only* variable recursively defines the set of strings of terminals that **S** can generate

- A CFG with more than one variable is a simultaneous recursive definition of the sets of strings generated by *each* of its variables
  - sometimes necessary to use more than one

CFGs and regular expressions

Theorem: For any set of strings (language) A described by a regular expression, there is a CFG that recognizes A.

Proof idea:

P(A) is “A is recognized by some CFG”

Structural induction based on the recursive definition of regular expressions…

Regular Expressions over Σ

- **Basis:**
  - ε is a regular expression
  - ***a*** is a regular expression for any ***a*** $\in \Sigma$
- **Recursive step:**
  - If **A** and **B** are regular expressions then so are:

    A $\cup$ B

    AB

    A*

CFGs are more general than REs

- CFG to match RE ε

S →

- CFG to match RE ***a*** (for any $a \in \Sigma$)

S → a

CFGs are more general than REs

Suppose CFG with start symbol $\mathbf{S_1}$ matches RE **A**
CFG with start symbol $\mathbf{S_2}$ matches RE **B**

- CFG to match RE $\mathbf{A} \cup \mathbf{B}$

  $\mathbf{S} \rightarrow \mathbf{S_1} \mid \mathbf{S_2}$ + rules from original CFGs

- CFG to match RE **AB**

  $\mathbf{S} \rightarrow \mathbf{S}_1\, \mathbf{S}_2$ + rules from original CFGs

CFGs are more general than REs

Suppose CFG with start symbol $\mathbf{S_1}$ matches RE **A**

- CFG to match RE $\mathbf{A}^*$ (= $\varepsilon \cup \mathbf{A} \cup \mathbf{AA} \cup \mathbf{AAA} \cup \ldots$)

  $\mathbf{S} \rightarrow \mathbf{S_1\,S} \mid \varepsilon$ + rules from CFG with $\mathbf{S_1}$

Backus-Naur Form (The same thing…)

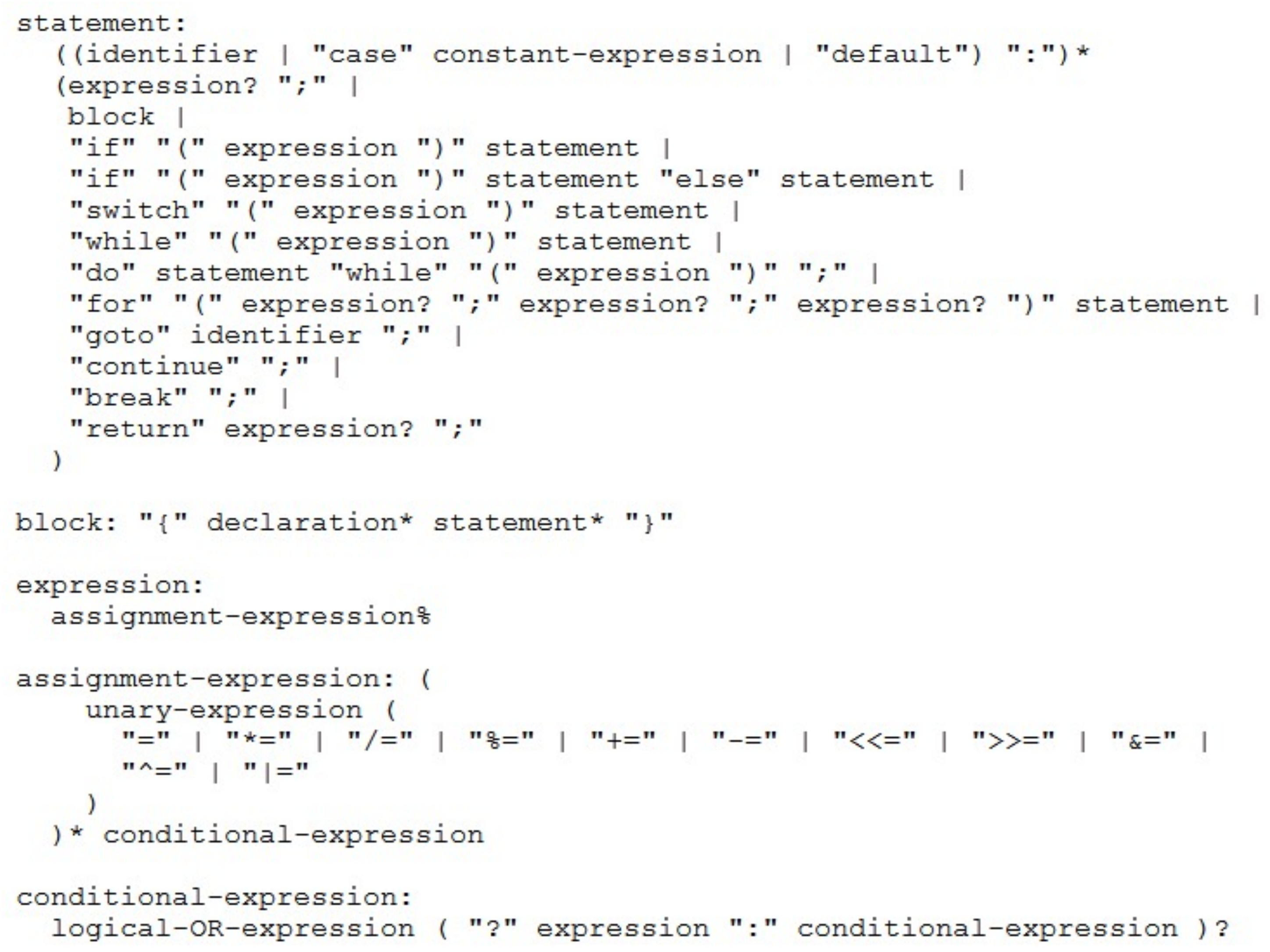
BNF (Backus-Naur Form) grammars

- Originally used to define programming languages
- Variables denoted by long names in angle brackets, e.g.

  <identifier>, <if-then-else-statement>,
  <assignment-statement>, <condition>

  $::=$ used instead of →

BNF for C



```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement "while" "(" expression ")" ";" |
   "for" "(" expression? ";" expression? ";" expression? ")" statement |
   "goto" identifier ";" |
   "continue" ";" |
   "break" ";" |
   "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
    unary-expression (
      "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
      "^=" | "|="
    )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?
```

BNF for (Simple) English

Back to middle school:

<sentence>::=<noun phrase><verb phrase>

<noun phrase>::==<article><adjective><noun>

<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

Parse:

The yellow duck squeaked loudly

The red truck hit a parked car