

# Grammars



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

# Outline and Objectives

## Outline

- Motivation - Parsing
- Context Free Grammars
  - Formal definition
  - Terminology
- Generating from a Grammar
- Parse trees and acceptance
  - Ambiguity
  - Look aheads

## Learning Objectives

- Describe parsing strategies for a context free grammar.
- Derive a parse tree given a grammar and string.
- Generate strings from a grammar.

# Parsing

```
while (expression) {  
    statement;  
    if (expression) {  
        statement;  
        statement;  
    }  
}
```

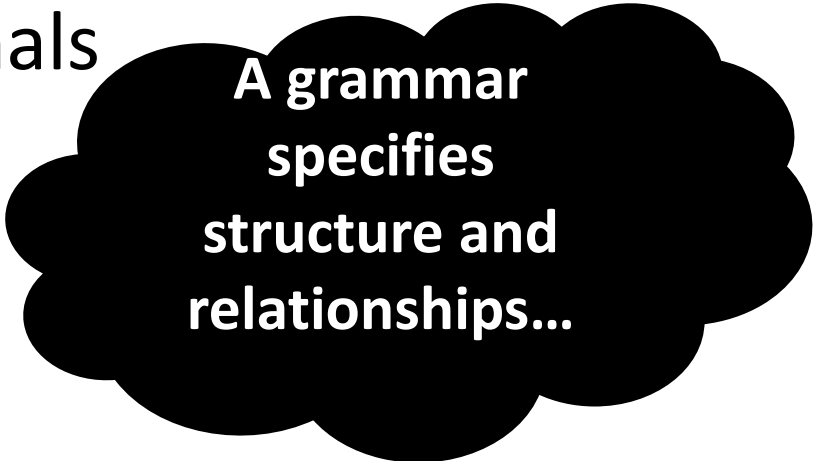
Is this code “valid”?

How can we formally represent or describe the structure of the code or the relationships between the different statements?

# Definition

Context-free grammar can be defined by:

- A finite set of variables that represent intermediate structures (variables) and terminals, called  $V$
- A set of rules,  $R$ , that relate variables to other variables and/or terminals
- A start variable  $S \in T$



**A grammar  
specifies  
structure and  
relationships...**

# Example

Let's assume we have the following grammar...

$$V = \{, \}$$

with the rule

$$S \rightarrow \{ \} \mid \{ S \} \mid SS$$

and  $S = S$

# Example

What can we derive from  
this grammar...

$$S \Rightarrow \{ S \}$$
$$\Rightarrow \{ \{ \} \}$$
$$S \Rightarrow SS$$
$$\Rightarrow \{ \{ S \} \}$$
$$\Rightarrow \{ \{ SS \} \}$$
$$\Rightarrow \{ \{ \{ \} \} \}$$

**To generate a string for  
the grammar, start from  
S and continue  
substituting...**

# Terminology

Variable – a named intermediary

Terminal – end product

Rule – a structural relationship between one variable and another.

Yields – application of one rule to a variable

Derives – a chain of rules exists to proceed from variable to another variable and/or terminal.





# Parsing

Revisit opening example...

Assume we have the following grammar...

$V = \mathbf{while}, ;, \mathbf{if}, (, ), \{, \}, \text{while\_statement},$   
 $\text{if\_statement}, \text{expression}, \text{statement}$

$S = \text{statement}$

# Parsing

$R$ , the substitution rules, are as follows

statement  $\rightarrow$  statement; statement |

while\_statement | if\_statement | expression |  $\epsilon$

while\_statement  $\rightarrow$  while ( expression ) { statement }

if\_statement  $\rightarrow$  if ( expression ) { statement }

# Parsing

```
while (expression) {  
    statement;  
    if (expression) {  
        statement;  
        statement;  
    }  
}
```

Tokenize!

while ( expression ) { statement ; if ( expression ) { statement ; statement ; } }

# Parsing

Next, we know we need to start at the start symbol, (ie, statement). From there, we scan across the stream of tokens...

[ Place image! ]

# Recursive Descent

This particular grammar has the advantage that we can tell by looking at the first token which rule we should apply!

This peek forward is called a 'look ahead'. If only one look ahead is needed the parsing can easily be handled by a recursive descent parser. Recursion to the rescue.

# Asked and Answered

- ✓ Since we have a parse tree for our snip of code, we know that it is valid and conforms to the grammar given.
- ✓ We also know how to formally represent the structure of the code (through the parse tree).

# Programming Language

Now you know why a programming language is called such ... it is specified by a grammar.

Technically, a programming language is comprised of all the valid programs that could be generated by the grammar that specifies it.

# References

- ❑ Sipser M: *Introduction to the theory of computation*. 2nd ed. Boston: Thomson Course Technology; 2006.
- ❑ Nisan N and Schocken S: *The Elements of Computing Systems*. Cambridge: MIT Press; 2006