CMSC 124
Design and Implementation of Programming Languages

Kristine Bernadette Pelaez
Institute of Computer Science
University of the Philippines Los Baños

Syntax Analysis

Syntax Analysis

aka parsing;
identifies the larger program structures
using the output of the scanner

Syntax Analysis

done by the
syntax analyzer
or parser

Syntax Analysis

once done,
it outputs a *parse tree*

Syntax Analysis

Parse Tree
describes the hierarchical && syntactic structure of the source program

Goals of Parsing

1. Determine if the program is syntactically correct.
2. Produce parse trees.

Syntax Analysis

parse trees follow the
grammar of the given language

Grammars

Grammars

language generation
mechanisms used
to *describe syntax*

Grammars

the widely used type
of grammar for describing syntax
are *context-free grammars*

Context-free Grammars

Context-free Grammars

aka *CFG*;
it is created by *Noam Chomsky*
and used to *describe recursive structures*

Context-free Grammars

Backus-Naur Form (BNF)
a notation for CFG used in the context
of PL specification and translation

Context-free Grammars

composed of
production rules

LHS ::= RHS

Context-free Grammars

LHS ::= RHS

the LHS contains the
abstraction being defined

Context-free Grammars

LHS ::= RHS

the RHS contains the
definition of the LHS

Context-free Grammars

LHS ::= RHS1 | RHS2

an abstraction may have
multiple definitions
separated by a *pipe* (|)

Context-free Grammars

grammars always have a *start variable*, usually the *LHS of the first production rule*

```
A ::= RHS1
B ::= RHS2
C ::= RHS3
```


Context-free Grammars

```
<NUMBER> ::= <DIGIT> | <NUMBER><DIGIT>
<DIGIT>  ::= 0 | 1 | 2 | 3 | 4 | 5
<DIGIT>  ::= 6 | 7 | 8 | 9
```


Context-free Grammars

```
<NUMBER> ::= <DIGIT> | <NUMBER><DIGIT>
<DIGIT>  ::= 0 | 1 | 2 | 3 | 4 | 5 |
             6 | 7 | 8 | 9
```


Context-free Grammars

the *language is generated* using its grammar *by repeatedly applying the rules*

Context-free Grammars

Derivation

sequence of repeated rule applications starting from the start variable

Context-free Grammars

in the case of the syntax analyzer, a *statement is syntactically correct if it can be derived* from the grammar of the language.

Two (2) Types of Derivation

1. Rightmost derivation
2. Leftmost derivation

Context-free Grammars

```
<NUMBER> ::= <DIGIT> | <NUMBER><DIGIT>
<DIGIT>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```


Context-free Grammars

each derivation can be *represented by a parse tree*

Parse Tree

visual representation
of a derivation


```
<NUMBER>
  ::= <DIGIT> |
      <NUMBER><DIGIT>

<DIGIT>
  ::= 0 | 1 | 2 | 3 |
      4 | 5 | 6 | 7 |
      8 | 9
```



```
 <expr> ::= <expr> <op> <expr> | <id> | <num>
   <op> ::= + | - | * | /
   <id> ::= A | B | C
  <num> ::= <num><digit> | <digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```


Ambiguity

a grammar is ambiguous
if it can *generate two or more distinct parse trees from the same statement*

DANGLING ELSE

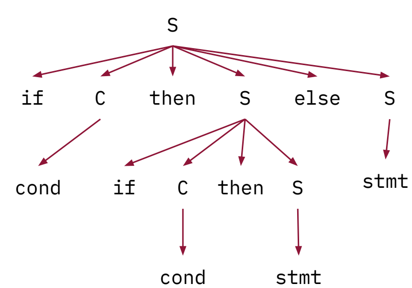
```
S ::= if C then S |
      if C then S else S |
      stmt
C ::= cond
```



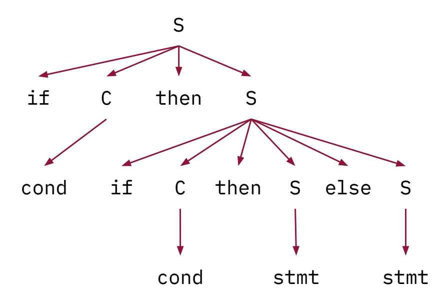
```
if cond then if cond then stmt else stmt
```



```
if cond then
   if cond then
      stmt
else
   stmt
```




```
if cond then
   if cond then
      stmt
   else
      stmt
```



Ambiguity

to remove *ambiguity*,
the grammar must be *rewritten*


```
E ::= E + E | E * E | (E) | A | B | C
```

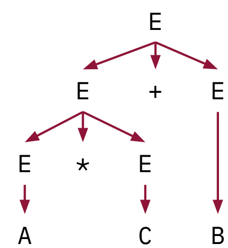

Operator Precedence

Operator Precedence

the *order* in which
operations will be evaluated

Operator Precedence

operations *lower*
in the parse tree
has higher precedence
and are *executed first*




```
E ::= E + E | E * E | (E) | A | B | C
```



```
E ::= E + E | T
T ::= T * T | P
P ::= (E) | V
V ::= A | B | C
```


Operator Associativity

Operator Associativity

specifies *which operation should be evaluated first* if the operations have the *same precedence*

Operator Associativity

```
A + B + C
  = (A + B) + C
  = A + (B + C)
```


Two (2) Types of Associativity

1. Left-associativity
2. Right-associativity

Operator Associativity

Left Associativity

evaluation if from left to right

Right Associativity

evaluation is from right to left

Operator Associativity

Left Associativity

```
A + B + C
A - B - C
A + B - C
A / B * C
```

Right Associativity

```
A^B^C^D^E
```


Operator Associativity

Left Associativity

left-recursive

```
A ::= AR | A
R ::= RA | RB
```

Right Associativity

right-recursive

```
A ::= RA | A
R ::= AR | BR
```



```
E ::= E + E | T
T ::= T * T | P
P ::= (E) | V
V ::= A | B | C
```



```
E ::= E + T | T
T ::= T * P | P
P ::= (E) | V
V ::= A | B | C
```


Syntax Analysis

the *grammar* of a language
is the basis of the parser
in checking for errors

Syntax Analysis

if a statement cannot be derived
using the grammar of the language,
then it does not follow its syntax

Two (2) Types of Parsers

1. Top-down parsers
2. Bottom-up parsers

Syntax Analysis

Top-down Parsers

Parse tree is created from the **root down to the leaves**.

Bottom-up Parsers

Parse tree is created from the **leaves up to the root**.

Syntax Analysis

Top-down Parsers

Uses **preorder traversal and leftmost derivations** in creating the parse tree.

Bottom-up Parsers

Uses **reverse rightmost derivations** in creating the parse tree.

Syntax Analysis

Top-down Parsers

Each abstraction is a function.

Syntax Analysis

S ::= S + T | T
T ::= (S) | id

CMSC 124

Design and Implementation of Programming Languages

Kristine Bernadette Pelaez
Institute of Computer Science
University of the Philippines Los Baños