Lecture 13

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/19/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- The Parsing Problem
- Top-Down Parsing
- Left-Recursion Removal (Review)
- Left Factoring (Review)
- Parsing PL/0
- Syntax Graphs

The Parsing Problem

We already know how to build *up to* statements and their associated parse trees *from* a grammar.

For a parser, we need to go the other direction. The parsing problem is nothing more or less than:

Given a grammar and a sentence in that grammar, produce a parse tree.

In the context of a compiler a sentence is a program, so:

Given a grammar and a program in that grammar, produce a parse tree.

There are actually two ways to do this. We will focus on *top-down* (or *recursive descent*) parsing, as opposed to *bottom-up* parsing.

Recursive Descent Parsing

In recursive descent, we:

- Begin at the top of the tree – the start symbol
  - In the case of a parser, that is the program itself
- Model the parse tree from the root on down
- Construct a left most derivation

Consider the grammar to the right.

- We’ve already removed all the problematic elements from it
  - (We’ll get back to those in a bit)
- This means that we can actually construct pseudocode to parse it in a very straightforward manner

E	::= T E’
E’	::= **“+”** T E’ | **ε**
T	::= F T’
T’	::= **“*”** F T’ | **ε**
F	::= **“(“** E **“)”** | **id**

Recursive Descent Parsing: Pseudocode 1

```
Procedure E
   begin { E }
      call T
      call E'
      print (" E found ")
   end { E }

Procedure E'
   begin { E' }
      If token = "+" then
        begin { IF }
          print (" + found ")
          Get next token
          call T
          call E'
        end { IF }
        print (" E' found ")
   end { E' }
```

E ::= T E’

E’ ::= **“+”** T E’ | **e**

T ::= F T’

T’ ::= **“*”** F T’ | **e**

F ::= **“(“** E **“)”** | **id**

Recursive Descent Parsing: Pseudocode 2

```
Procedure T
   begin { T }
      call F
      call T’
      print (“ T found ”)
   end { T }

Procedure T’
   begin { T’ }
      If token = “ * ” then
        begin { IF }
          print (“ * found “)
          Get next token
          call F
          call T’
        end { IF }
        print (“ T’ found “)
   end { T’ }
```

E	::= T E’
E’	::= **“+”** T E’ | **e**
T	::= F T’
T’	::= **“*”** F T’ | **e**
F	::= **“(“** E **“)”** | **id**

Recursive Descent Parsing: Pseudocode 3

```
Procedure F
  begin { F }
    case token is
    "(":
        print (" (  found ")
        Get next token
        call E
        if token = ")" then
          begin { IF }
            print (" ) found")
            Get next token
             print (" F found ")
          end { IF }
        else
        call ERROR
    "id":
        print (" id found ")
        Get next token
        print (" F found ")
   otherwise:
        call ERROR
  end { F }
```

E ::= T E'

E' ::= "+" T E' | **e**

T ::= F T'

T' ::= "*" F T' | **e**

F ::= "(" E ")" | **id**

Left-Recursion (Review)

A grammar is *left-recursive* if it has a non-terminal symbol that can be written to itself followed by something else – that is, if it has any derivation of the form $A ::= A\ \alpha$

- Sadly, most grammars useful for anything are left-recursive in their simplest forms
- Top-down parsers can't easily handle left-recursive grammars
- We have to transform the grammar to eliminate left-recursion

For example, we could rewrite $A ::= A\ \alpha \mid \beta$ as:

$$A ::= \beta\ A'$$

$$A' ::= \alpha\ A' \mid \varepsilon$$

Left-Recursion Example (Review)

E ::= E “+” E | T

T ::= T “*” F | F

F ::= id | “(“ E “)”

The key is always the same: find a way to move the iteration to the right instead of the left.

What you’ll usually do is create an intermediate rule that resolves to either a rightward expansion of itself or the empty string.

E ::= T E’

E’ ::= “+” T E’ | ε

T ::= F T’

T’ ::= “*” F T’ | ε

F ::= “(“ E “)” | id

Left Factoring (Review)

We also want to avoid multiple rewrite rules whose substitution side starts the same way – and unfortunately, rules created by the OR operator count. So this…

$$A ::= \alpha\ \beta_1 \mid \alpha\ \beta_2$$

…is a problem. The good news is, it's easy to get rid of:

$$A ::= \alpha\ A'$$

$$A' ::= \beta_1 \mid \beta_2$$

…and that approach readily extends.

EBNF for PL/0 – Part 1

```
program               ::= block "."
block                 ::= const-declaration var-declaration proc-declaration statement
const-declaration     ::= [ "const" ident "=" number {"," ident "=" number} ";"]
var-declaration   ::=     [ "var" ident {"," ident} “;"]
proc-declaration  ::=     { "procedure" ident ";" block ";" }
statement             ::= [ ident ":=" expression
                            | "call" ident
                            | "begin" statement { ";" statement } "end"
                            | "if" condition "then" statement [“else" statement]
                            | "while" condition "do" statement
                            | "read" ident
                            | "write" ident
                            | e ]
```

EBNF for PL/0 – Part 2

```
condition     ::= "odd" expression
                | expression rel-op expression
rel-op        ::= "=" | “<>" | "<" | "<=" | ">" | ">="
expression    ::= [ "+"|"-"] term { ("+"|"-") term}
term          ::= factor {("*"|"/") factor}
factor        ::= ident | number | "(" expression ")"
number        ::= digit {digit}
ident         ::= letter {letter | digit}
digit         ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
letter        ::= "a" | "b" | … | "y" | "z" | "A" | "B" | ... | "Y" | "Z"
```

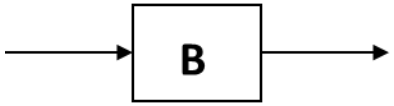
A Better Way

- Building a parser straight out of the EBNF is, as we have just seen, entirely possible
- It’s also a little bit of a mess
- A better – or at least more intuitive – way is to turn it into a *graph* first

Syntax Graphs: Basics

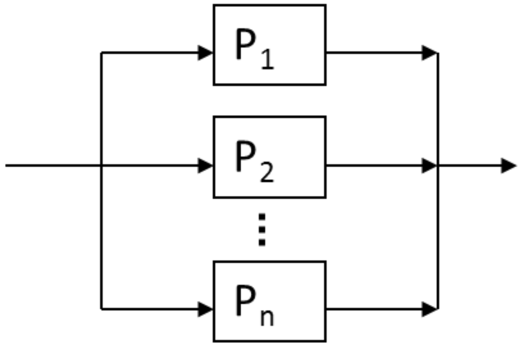
Every occurrence of a terminal symbol in a production means that a token has been recognized and a new symbol (token) must be read. This is represented by a label enclosed in a circle.

Every occurrence of a non-terminal symbol in a production corresponds to an activation of its recognizer.

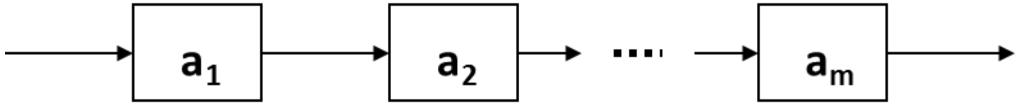


Syntax Graphs: Alternation and Concatenation

$A ::= P_1 \mid P_2 \mid . . . \mid P_n$

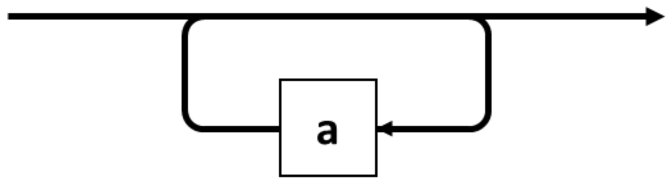


$P ::= a_1\, a_2\, . . .\, a_m$

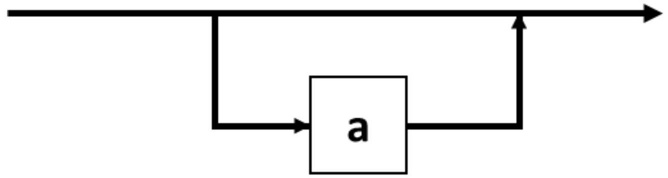


Syntax Graphs:
Closure and Optional Items

P = {a}



P = [a]

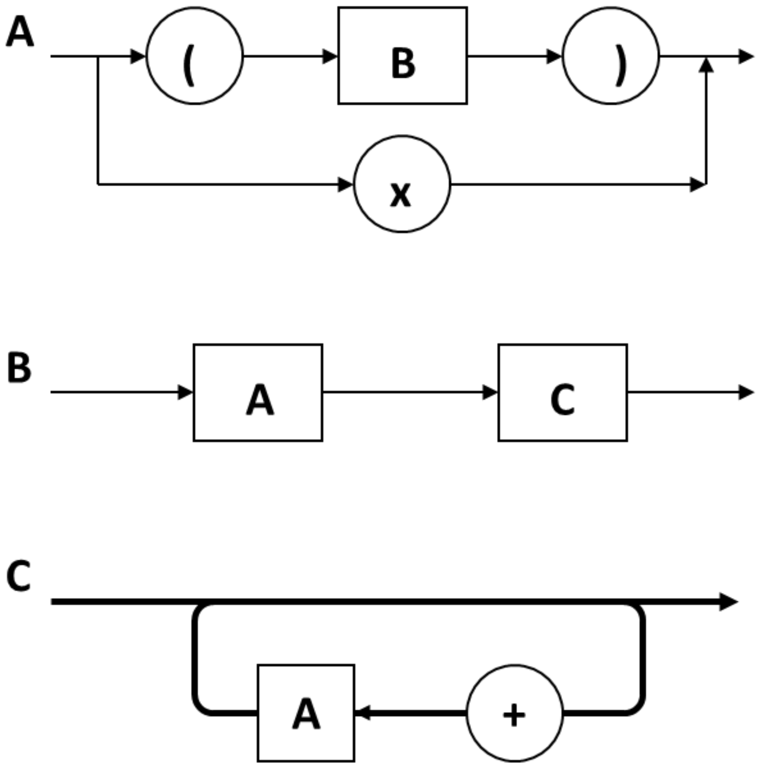


Syntax Graph Example
(from N. Wirth himself, no less)

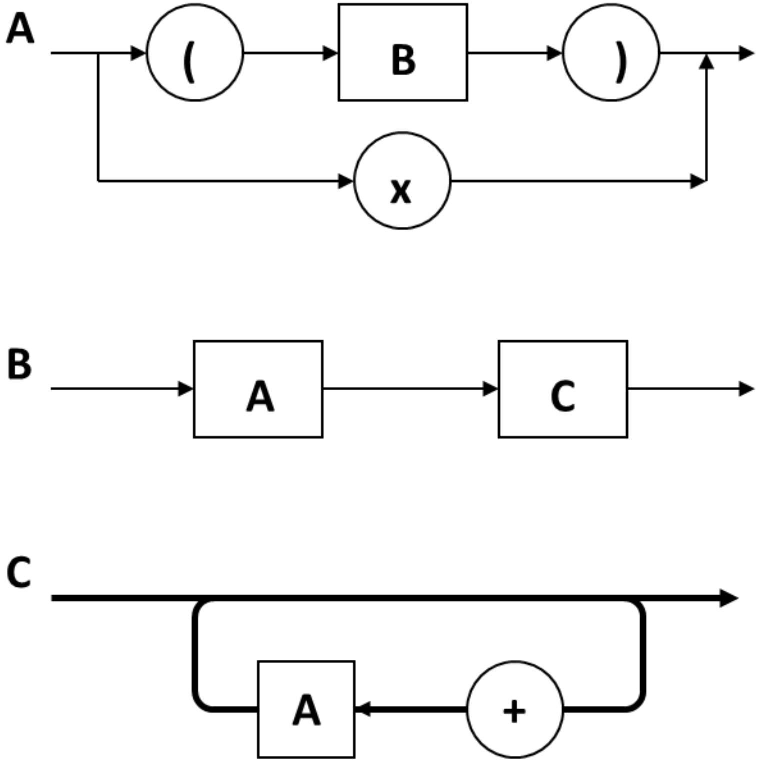
A ::= “x” | “(“ B “)”

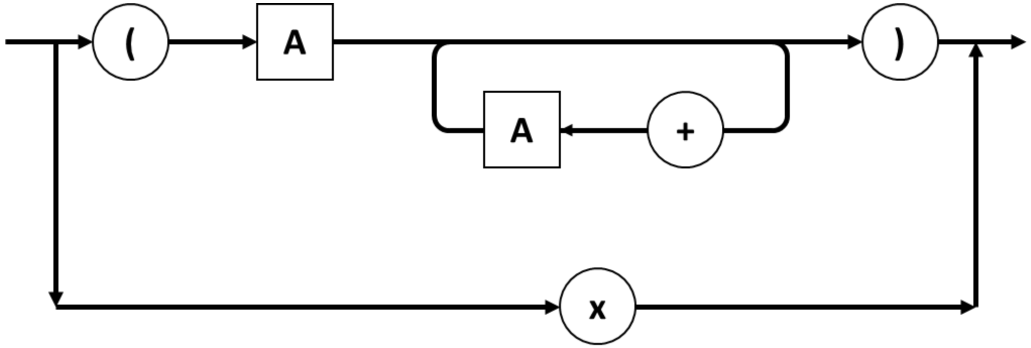
B ::= A C

C ::= { “+” A }



Syntax Graph Example – Fully Composed





Next Time:
Building a Parser