COP 3402 Systems Software

Top Down Parsing
(Recursive Descent)

Outline

1. Top down parsing and LL(k) parsing
2. Recursive descent parsing
3. Example of recursive descent parsing of arithmetic expression
   a) based on predictive parsing table
   b) based on syntax graph
4. Extended Backus-Naur form
5. Syntax graph
6. A LL(1) grammar for PL/0 in extended Backus-Naur form

Top-Down Parsing

A top-down parser is a parser that build the parse tree by starting at the root and working down the parse tree.

LL Parsing

LL parsing is a top-down parser that parses input **L**eft to right performing **L**eftmost derivations.

LL(k) Parsing

An LL parser is called an LL(k) parser if it uses k tokens of look-ahead to choose the correct productions when parsing sentences (without ever having to backtrack).

If such a parser exists for a certain grammar, then this grammar is called an LL(k) grammar.

We will construct a LL(1) parser for parsing PL/0 programs.

Recursive Descent Parsing

Recursive Descent Parsing uses recursive functions to model the parse tree to be constructed.

The parse tree is built from the top down, trying to construct a left-most derivation.

For each non-terminal symbol of the grammar (syntactic class), we have to implement a function that parses it.

Recursive Descent Parsing

We want to implement a recursive descent parser for the following LL(1) grammar for arithmetic expressions:

```
E  -> T E'
E' -> + T E' | ε
T  -> F T'
T' -> * F T' | ε
F  -> ( E ) | id | num
```

Predictive Parsing Table

```
S -> E $
E -> T E'
E'-> + T E' | ε
T -> F T'
T'-> * F T' | ε
F -> ( E ) | id | num
```

We will cover later how to construct the predictive parsing table.

The column for `num` is omitted because it is the same as for `id`

	+	*	id	(	)	$
S			S -> E $	S -> E $		
E			E -> T E'	E -> T E'		
E'	E'-> + T E'				E'-> ε	E'-> ε
T			T -> F T'	T -> F T'		
T'	T'-> ε	T'-> * F T'			T'-> ε	T'-> ε
F			F -> id	F -> (E)		

Predictive Parsing Table

The corresponding code is in the file

```
predictive-table-parser.c
```

in the folder

```
2_parser/b_simple_parser
```

Extended Backus-Naur Form

```
E  -> T E'
E' -> + T E' | ε
T  -> F T'
T' -> * F T' | ε
F  -> ( E ) | id | num
```

```
E  -> T E'
E' -> { + T }
T  -> F T'
T' -> { * F }
F  -> ( E ) | id | num
```

Note that { and } are not terminal symbols of the grammar.

They are symbols of the EBNF used to indicate that something can occur an arbitrary number of times (0, 1, 2, …).

Note that we could eliminate E' and T' as follows:

```
E  -> T { + T }
T  -> F { * F }
```

Extended Backus-Naur Form

```
E  -> T E'
E' -> { + T }
T  -> F T'
T' -> { * F }
F  -> ( E ) | id | num
```

The code based on the above EBNF specification is shown in pseudo code on the following slide.

An implementation in C is in the file

`syntax-graph-parser.c`

in the folder

`2_parser/b_simple_parser`

Recursive Descent Parsing

```
procedure E
begin
  call T
  call E'
end { E }

procedure E'
begin { E' }
  while token = "+"
  begin { WHILE }
    Get next token
    call T
  end { WHILE }
end { E' }
```

```
procedure T
begin { T }
  call F
  call T'
end { T }

procedure T'
begin { T' }
  while token = "*"
  begin { WHILE }
    Get next token
    call F
  end { IF }
end { T' }
```

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

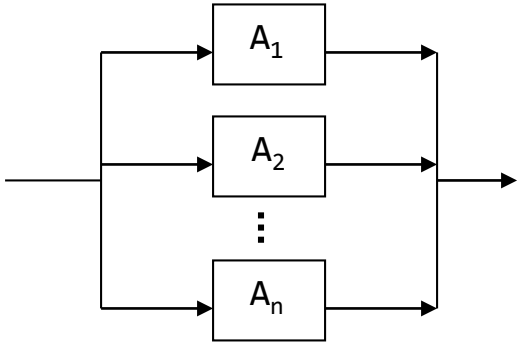
Syntax Graph

It is advantageous to transform a grammar expressed in EBNF into a syntax graph.

This helps visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

Rules to construct a syntax graph:

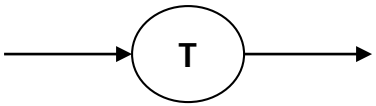
R1.- Alternation $A_1 \mid A_2 \mid \ldots \mid A_n$ is represented by the following syntax graph:



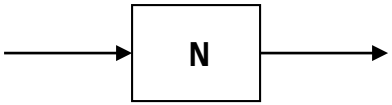
Syntax Graph

Rules to construct a syntax graph:

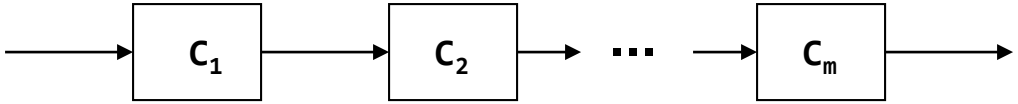
R2.- Every occurrence of a *terminal symbol* `T` means that a token has been recognized and a new symbol (token) must be read. This is represented by a label `T` enclosed in a circle.



R3.- Every occurrence of a *non-terminal symbol* `N` corresponds to an activation of the recognizer `N`.



R4.- Concatenation $C_1\ C_2\ \ldots\ C_m$ is represented by the graph:

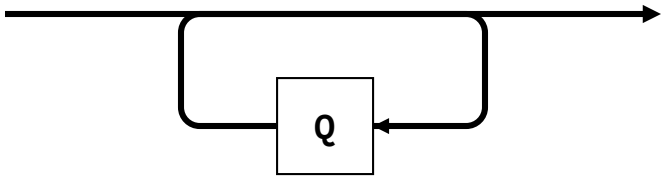


where every C_i is obtained by applying construction rules R1 through R6.

Syntax Graph

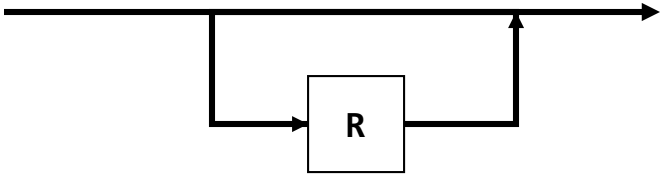
Rules to construct a syntax graph:

R5.- { Q } is represented by the graph:



where Q is obtained by applying constructing rules R1 through R6

R6.- [R] is represented by the graph:

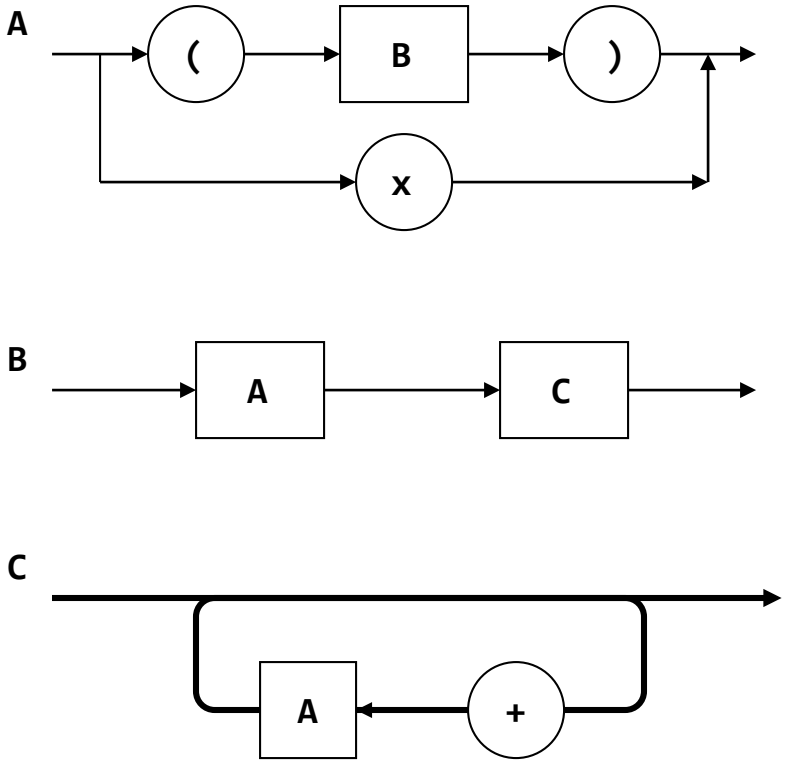


where R is obtained by applying rules R1 through R6

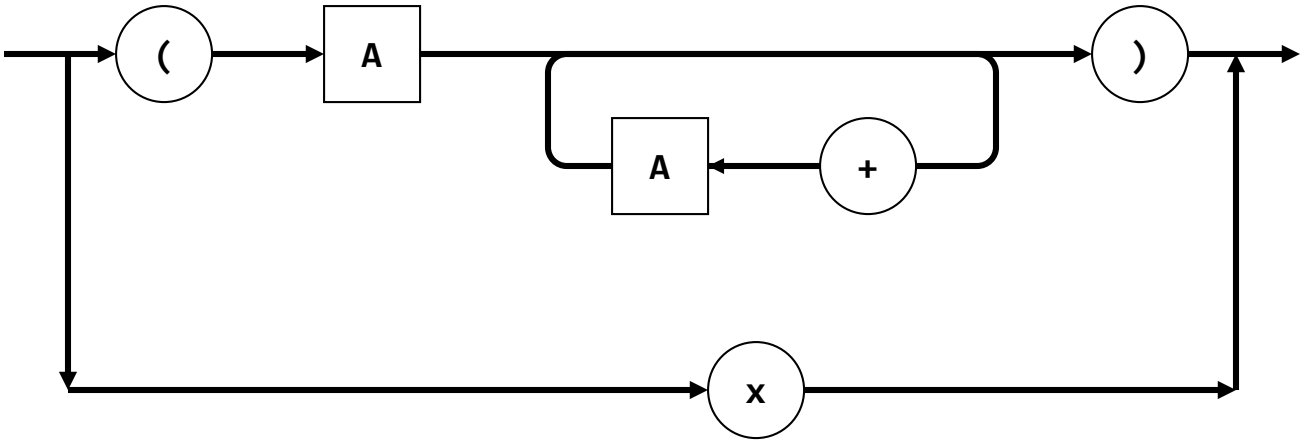
Syntax Graph

Example due Nikolas Wirth:

```
A ::= “x” | “(” B “)”
B ::= A C
C ::= { “+” A }
```



Syntax Graph



This is the final syntax graph.

Extended BNF Grammar for PL/0

```
<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
                            | “while” <condition> “do” <statement>
                        ]

<condition>         ::=   “odd” <expression>
                        | <expression> <rel-op> <expression>

<rel-op>            ::= “=” | “<>” | “<” | “<=” | “>” | “>=”
```

Extended BNF Grammar for PL/0

<expression> ::= [“+” | “-”] <term> { (“+” |“-”) <term> }

<term> ::= <factor> { (“*“ | “/”) <factor>}

<factor> ::= <ident> | <number> | “(” <expression> “)”

Note that <ident> and <number> should be considered as terminal symbols of the grammar/tokens for the parser.

<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”

Error Messages for PL/0 Parser

Error messages for the PL/0 Parser:

1. Use = instead of :=.
2. = must be followed by a number.
3. Identifier must be followed by =.
4. **const, var, procedure** must be followed by identifier.
5. Semicolon or comma missing.
6. Incorrect symbol after procedure declaration.
7. Statement expected.
8. Incorrect symbol after statement part in block.
9. Period expected.
10. Semicolon between statements missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
13. Assignment operator expected.
14. **call** must be followed by an identifier.
15. Call of a constant or variable is meaningless.
16. **then** expected.
17. Semicolon or **end** expected.
18. **do** expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot begin with this symbol.
24. An expression cannot begin with this symbol.
25. This number is too large.