COP 3402 Systems Software

Recursive Descent Parsing

Outline

1. The parsing problem
2. Top-down parsing
3. Left-recursion removal
4. Left factoring
5. EBNF grammar for PL/O

Recursive descent parsing

The parsing Problem: Take a string of symbols in a language (tokens) and a grammar for that language to construct the parse tree or report that the sentence is syntactically incorrect.

For correct strings:

Sentence + grammar → parse tree

For a compiler, a sentence is a program:

Program + grammar → parse tree

Types of parsers:

Top-down (recursive descent parsing)

Bottom-up parsing.

Recursive descent parsing

Recursive Descent parsing uses recursive procedures to model the parse tree to be constructed. The parse tree is built from the top down, trying to construct a left-most derivation.

Beginning with ***start*** symbol, for each non-terminal (syntactic class) in the grammar a procedure which parses that syntactic class is constructed.

Consider the expression grammar:

E → T E'
E' → **+** T E' | **ε**
T → F T'
T' → ***** F T' | **ε**
F → **(** E **)** | **id**

The following procedures have to be written:

Recursive descent parsing

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' }
```

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' }
```

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 }
```

Error messages

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 most 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.

Recursive descent parsing

Ambiguity if not the only problem associated with recursive descent parsing. Other problems to be aware of are left recursion and left factoring:

Left recursion: A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \rightarrow A\,\alpha$ for some string α. Top-down parsing methods can not handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

For example, the pair of productions: $A \rightarrow A\,\alpha \mid \beta$

could be replaced by the non-left-recursive productions: $A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$

Recursive descent parsing

Left factoring: Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive (top-down) parsing. When the choice between two alternative A-production is not clear, we may be able to rewrite the production to defer the decision until enough of the input has been seen thus we can make the right choice.

For example, the pair of productions: $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$

could be left-factored to the following productions: $A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

Extended BNF grammar for PL/0 (1)

<program> ::= <block> "**.**" .

<block> ::= <const-declaration> <var-declaration> <procedure-declaration> <statement>

<constdeclaration> ::= [“**const**” <ident> "**=**" <number> {"**,**" <ident> "**=**" <number>} "**;**"]

<var-declaration> ::= ["**var**" <ident> {"**,**” < ident> } "**;**"]

<procedure-declaration> ::= { "**procedure**" <ident> "**;**" <block> "**;**" }

<statement > ::= [<ident> "**:=**" <expression>
| "**call**" <ident>
| "**begin**" <statement> {"**;**" <statement> } "**end**"
| "**if**" <condition> "**then**" statement
| "**while**" <condition> "**do**" <statement>
| ε]

<condition> ::= "**odd**" <expression>
| <expression> <rel-op> <expression>

<rel-op> ::= "**=**"|“**<>**"|"**<**"|"**<=**"|"**>**"|"**>=**“

Extended BNF grammar for PL/0 (2)

<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"

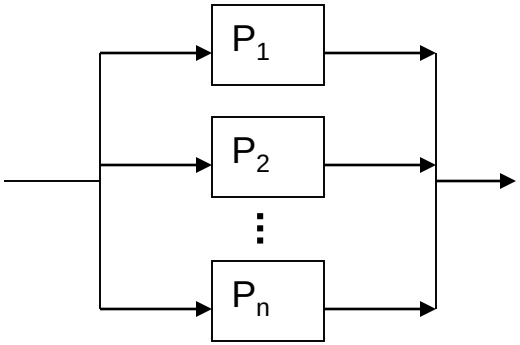
Syntax Graph

Transforming a grammar expressed in EBNF to syntax graph is advantageous to 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.- Each non-terminal symbol A which can be expressed as a set of productions

A ::= P_1 | P_2 | . . . | P_n can be mapped into the following syntax graph:

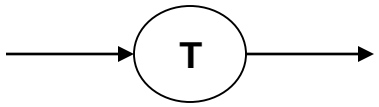


Syntax Graph

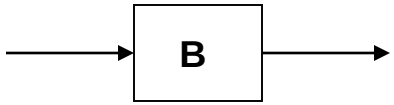
Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

Rules to construct a syntax graph:

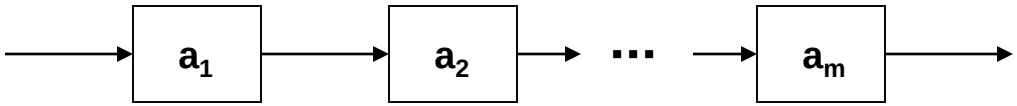
R2.- Every occurrence of a terminal symbol T in a P_i means that a token has been recognized and a new symbol (token) must be read. This is represented by a label T enclosed in a circle.



R3.- Every occurrence of a non-terminal symbol B in a P_i corresponds to an activation of the recognizer B.



R4.- A production P having the form $P = a_1\, a_2 \ldots a_m$ can be represented by the graph:



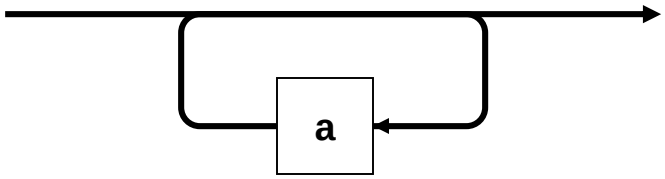
where every a_i is obtained by applying construction rules **R1** through **R6**

Syntax Graph

Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

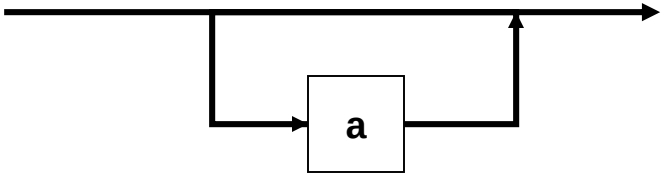
Rules to construct a syntax graph:

R5.- A production P having the form P = {a} can be represented by the graph:



where **a** is obtained by applying constructing rules R1 through R6

R6.- A production P having the form P = [a] can be represented by the graph:

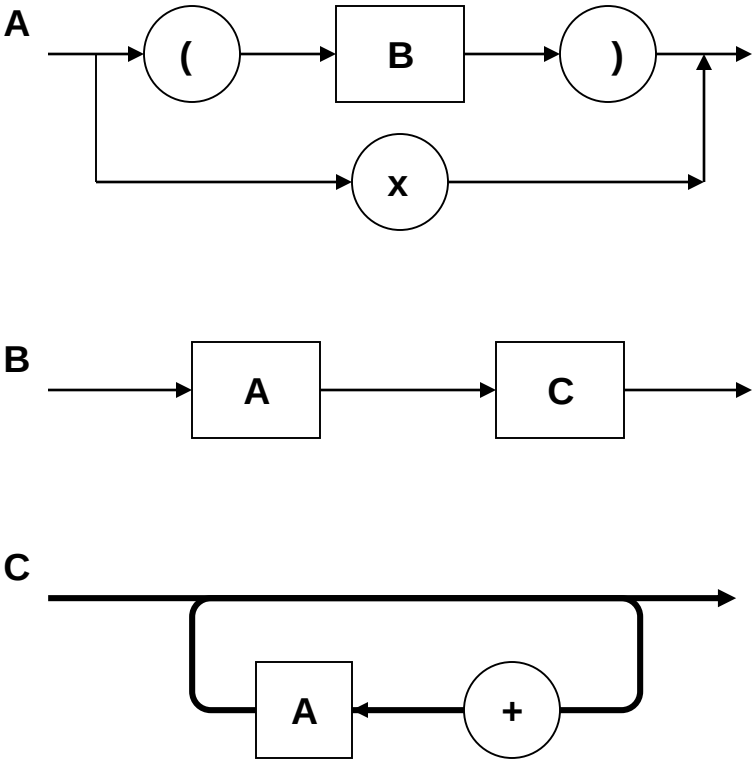


where **a** is obtained by applying constructing rules R1 through R6

Syntax Graph

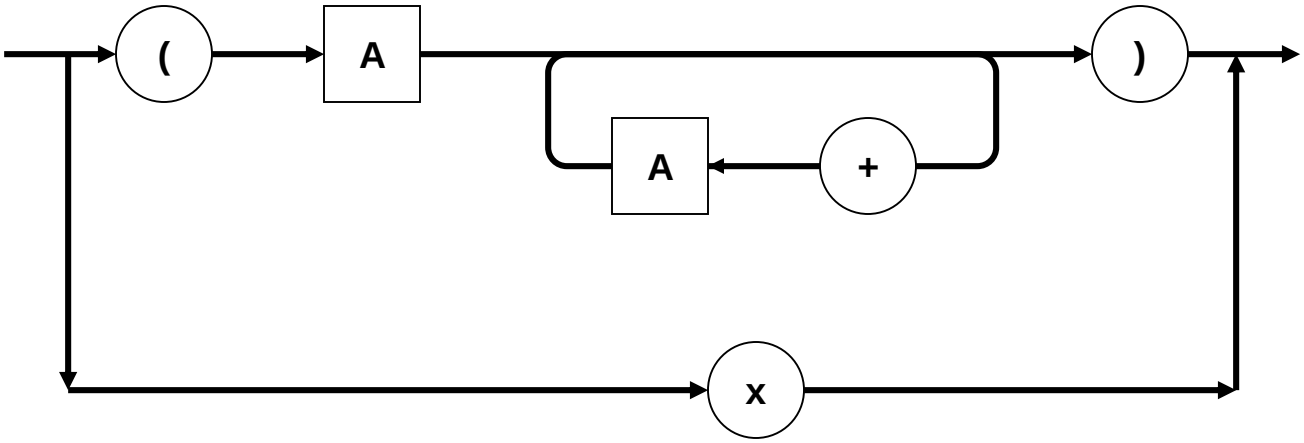
Example from N. Wirth:

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



x
(x)
(x + x)

Syntax Graph



This is the final syntax graph corresponding to Example 5 after

THE END