



ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

# Unit 4

## BNF and Syntax Diagrams

# Backus-Naur form and variants

- **Metasyntax**: a syntax used to describe the syntax of languages,
- **BNF** (Backus–Naur Form) is a metasyntax used to express context free grammars
- BNF is widely used as a notation for the grammars programming languages, instruction sets, communication protocols and parts of natural language grammars

# Backus-Naur form and variants (cont)

- A set of rules is specified. These are known as **production** rules.
- Each production rule defines the pattern that represents a named structured part of the language
- The name of such a part is called a **non-terminal** symbol in the language.
- The basic elements of the language are called **terminal** symbols.

# Backus-Naur form and variants (cont)

- Each rule contains the name of the non-terminal being defined, followed by the sequence or alternative sequences allowed for that symbol. A defining sequence can contain any terminal and non-terminal symbols allowed for that language.
- The definition of a rule can also contain the symbol being defined by that rules. This is called **recursive** definition.

# Example: Grammar for Arithmetic Expressions

- **Productions**

$\langle \text{Exp} \rangle ::= "+" \langle \text{Expr2} \rangle \mid "-" \langle \text{Expr2} \rangle \mid \langle \text{Expr2} \rangle$   
 $\langle \text{Expr2} \rangle ::= \langle \text{Term} \rangle \langle \text{Expr3} \rangle$   
 $\langle \text{Expr3} \rangle ::= "+" \langle \text{Term} \rangle \langle \text{Expr3} \rangle \mid$   
 $\quad "-" \langle \text{Term} \rangle \langle \text{Expr3} \rangle \mid \varepsilon$   
 $\langle \text{Term} \rangle ::= \langle \text{Factor} \rangle \langle \text{Term2} \rangle$   
 $\langle \text{Term2} \rangle ::= "*" \langle \text{Factor} \rangle \langle \text{Term2} \rangle \mid "/" \langle \text{Factor} \rangle \langle \text{Term2} \rangle \mid \varepsilon$   
 $\langle \text{Factor} \rangle ::= \text{"ident"} \mid \text{"number"} \mid "(" \langle \text{Exp} \rangle ")"$

- **Terminal symbols**

- simple TS: "+", "-", "\*", "/", "(", ")"
- terminal classes: "ident", "number"

- **Nonterminal symbols**

- $\langle \text{Expr} \rangle$ ,  $\langle \text{Expr2} \rangle$ ,  $\langle \text{Expr3} \rangle$ ,  $\langle \text{Term} \rangle$ ,  $\langle \text{Term2} \rangle$ ,  $\langle \text{Factor} \rangle$

- **Start symbol**

- $\langle \text{Exp} \rangle$

# EBNF(Extended BNF)

- Terminal symbols start with lower-case letters
- Nonterminal symbols start with upper-case letters
- **Metasymbols**
  - | (...) separates alternatives groups
  - [...] alternatives optional part
  - {...} iterative part

# KPL Grammar in BNF

01) Prog ::= KW\_PROGRAM Ident SB\_SEMICOLON Block SB\_PERIOD

02) Block ::= KW\_CONST ConstDecl ConstDecls Block2

03) Block ::= Block2

04) Block2 ::= KW\_TYPE TypeDecl TypeDecls Block3

05) Block2 ::= Block3

06) Block3 ::= KW\_VAR VarDecl VarDecls Block4

07) Block3 ::= Block4

08) Block4 ::= SubDecls Block5

09) Block5 ::= KW\_BEGIN Statements KW\_END

# KPL Grammar in BNF

- 10) `ConstDecls ::= ConstDecl ConstDecls`
- 11) `ConstDecls ::=  $\epsilon$`
- 12) `ConstDecl ::= Ident SB_EQUAL Constant SB_SEMICOLON`
- 13) `TypeDecls ::= TypeDecl TypeDecls`
- 14) `TypeDecls ::=  $\epsilon$`
- 15) `TypeDecl ::= Ident SB_EQUAL Type SB_SEMICOLON`
- 16) `VarDecls ::= VarDecl VarDecls`
- 17) `VarDecls ::=  $\epsilon$`
- 18) `VarDecl ::= Ident SB_COLON Type SB_SEMICOLON`
- 19) `SubDecls ::= FunDecl SubDecls`
- 20) `SubDecls ::= ProcDecl SubDecls`
- 21) `SubDecls ::=  $\epsilon$`



# KPL Grammar in BNF

22) FunDecl ::= KW\_FUNCTION Ident Params SB\_COLON BasicType  
SB\_SEMICOLON Block SB\_SEMICOLON

23) ProcDecl ::= KW\_PROCEDURE Ident Params SB\_SEMICOLON Block  
SB\_SEMICOLON

24) Params ::= SB\_LPAR Param Params2 SB\_RPAR

25) Params ::=  $\epsilon$

26) Params2 ::= SB\_SEMICOLON Param Params2

27) Params2 ::=  $\epsilon$

28) Param ::= Ident SB\_COLON BasicType

29) Param ::= KW\_VAR Ident SB\_COLON BasicType

# KPL Grammar in BNF

- 30) `Type ::= KW_INTEGER`
- 31) `Type ::= KW_CHAR`
- 32) `Type ::= TypeIdent`
- 33) `Type ::= KW_ARRAY SB_LSEL Number SB_RSEL KW_OF Type`
  
- 34) `BasicType ::= KW_INTEGER`
- 35) `BasicType ::= KW_CHAR`
  
- 36) `UnsignedConstant ::= Number`
- 37) `UnsignedConstant ::= ConstIdent`
- 38) `UnsignedConstant ::= ConstChar`
  
- 40) `Constant ::= SB_PLUS Constant2`
- 41) `Constant ::= SB_MINUS Constant2`
- 42) `Constant ::= Constant2`
- 43) `Constant ::= ConstChar`
  
- 44) `Constant2 ::= ConstIdent`
- 45) `Constant2 ::= Number`

# KPL Grammar in BNF

46) `Statements ::= Statement Statements2`

47) `Statements2 ::= KW_SEMICOLON Statement Statements2`

48) `Statements2 ::=  $\epsilon$`

49) `Statement ::= AssignSt`

50) `Statement ::= CallSt`

51) `Statement ::= GroupSt`

52) `Statement ::= IfSt`

53) `Statement ::= WhileSt`

54) `Statement ::= ForSt`

55) `Statement ::=  $\epsilon$`

# KPL Grammar in BNF

- 56) AssignSt ::= Variable SB\_ASSIGN Expression
- 57) AssignSt ::= FunctionIdent SB\_ASSIGN Expression
- 58) CallSt ::= KW\_CALL ProcedureIdent Arguments
- 59) GroupSt ::= KW\_BEGIN Statements KW\_END
- 60) IfSt ::= KW\_IF Condition KW\_THEN Statement ElseSt
- 61) ElseSt ::= KW\_ELSE Statement
- 62) ElseSt ::=  $\epsilon$
- 63) WhileSt ::= KW\_WHILE Condition KW\_DO Statement
- 64) ForSt ::= KW\_FOR VariableIdent SB\_ASSIGN Expression  
KW\_TO Expression KW\_DO Statement

# KPL Grammar in BNF

65) `Arguments ::= SB_LPAR Expression Arguments2 SB_RPAR`

66) `Arguments ::=  $\epsilon$`

67) `Arguments2 ::= SB_COMMA Expression Arguments2`

68) `Arguments2 ::=  $\epsilon$`

68) `Condition ::= Expression Condition2`

69) `Condition2 ::= SB_EQ Expression`

70) `Condition2 ::= SB_NEQ Expression`

71) `Condition2 ::= SB_LE Expression`

72) `Condition2 ::= SB_LT Expression`

73) `Condition2 ::= SB_GE Expression`

74) `Condition2 ::= SB_GT Expression`

# KPL Grammar in BNF

- 75) `Expression ::= SB_PLUS Expression2`
- 76) `Expression ::= SB_MINUS Expression2`
- 77) `Expression ::= Expression2`
  
- 78) `Expression2 ::= Term Expression3`
  
- 79) `Expression3 ::= SB_PLUS Term Expression3`
- 80) `Expression3 ::= SB_MINUS Term Expression3`
- 81) `Expression3 ::=  $\epsilon$`
  
- 82) `Term ::= Factor Term2`
  
- 83) `Term2 ::= SB_TIMES Factor Term2`
- 84) `Term2 ::= SB_SLASH Factor Term2`
- 85) `Term2 ::=  $\epsilon$`

# KPL Grammar in BNF

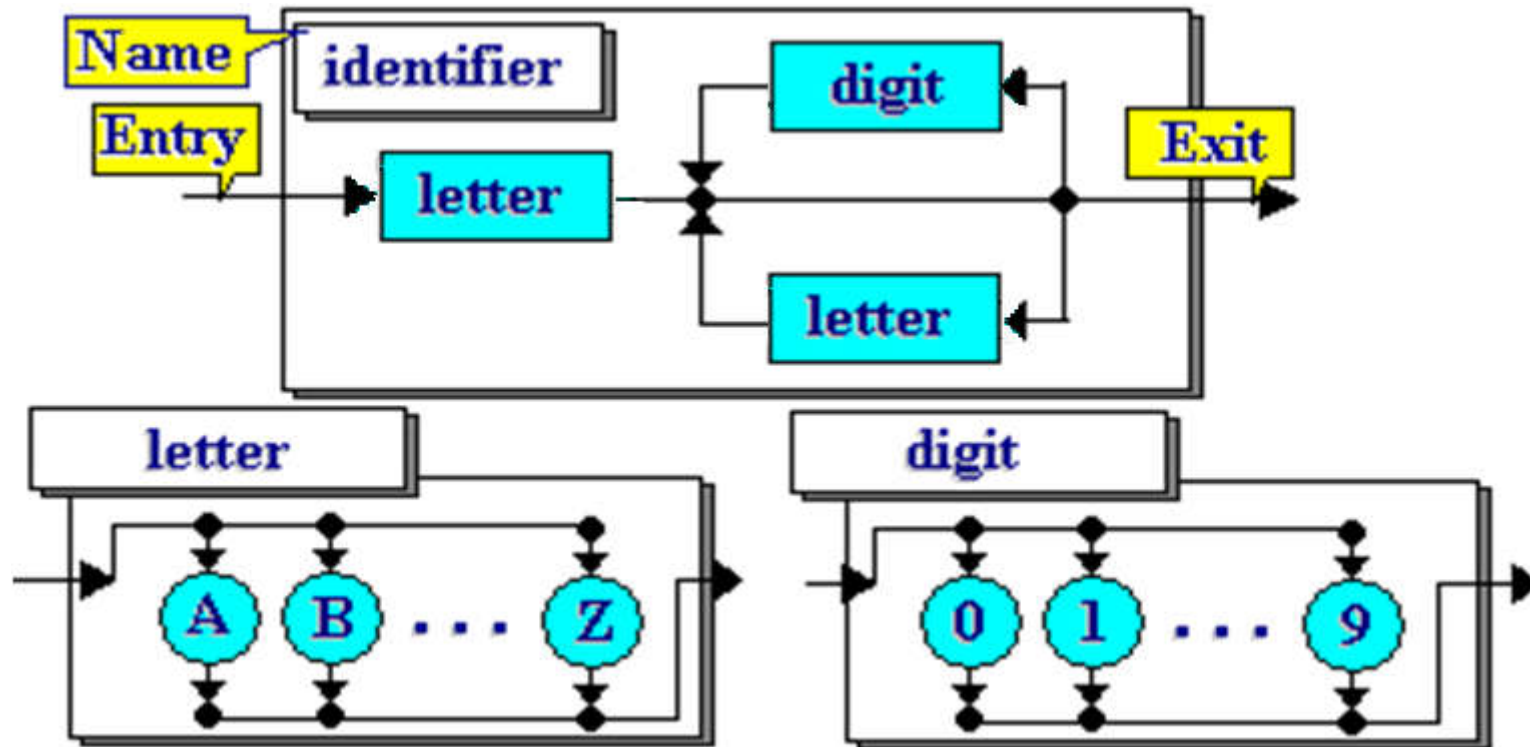
- 86) `Factor ::= UnsignedConstant`
- 87) `Factor ::= Variable`
- 88) `Factor ::= FunctionApplication`
- 89) `Factor ::= SB_LPAR Expression SB_RPAR`
  
- 90) `Variable ::= VariableIdent Indexes`
- 91) `FunctionApplication ::= FunctionIdent Arguments`
  
- 92) `Indexes ::= SB_LSEL Expression SB_RSEL Indexes`
- 93) `Indexes ::=  $\epsilon$`

# Syntax Diagram

- Each diagram defines a non-terminal
- There is a main diagram which defines the language
- Each diagram has an entry point and an end point
- Terminals are represented by round boxes
- Nonterminals are represented by square boxes.

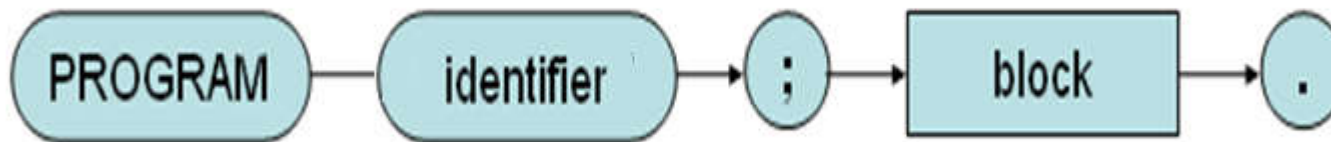


# Examples of syntax diagram

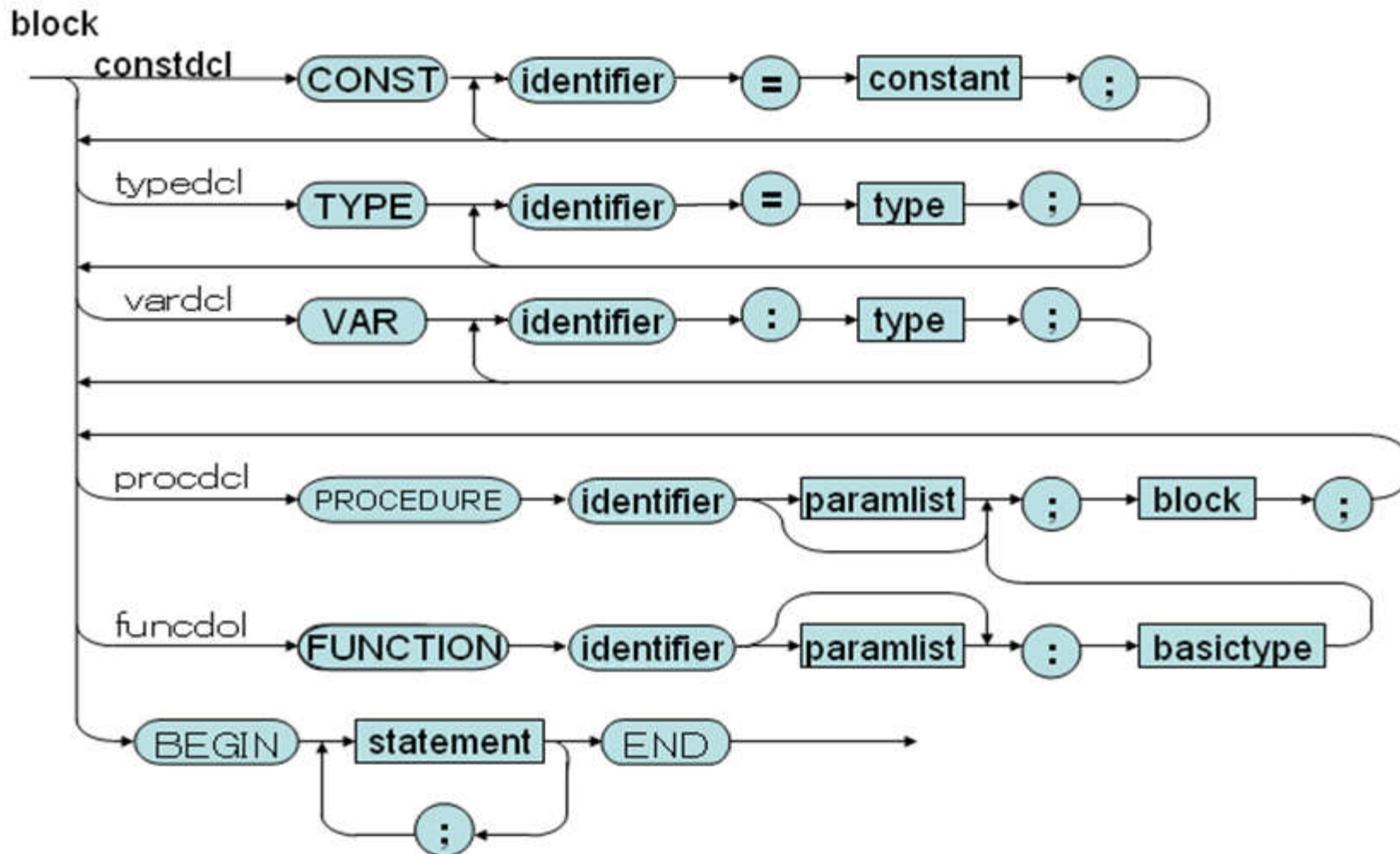


# Syntax Diagrams of KPL (program)

program

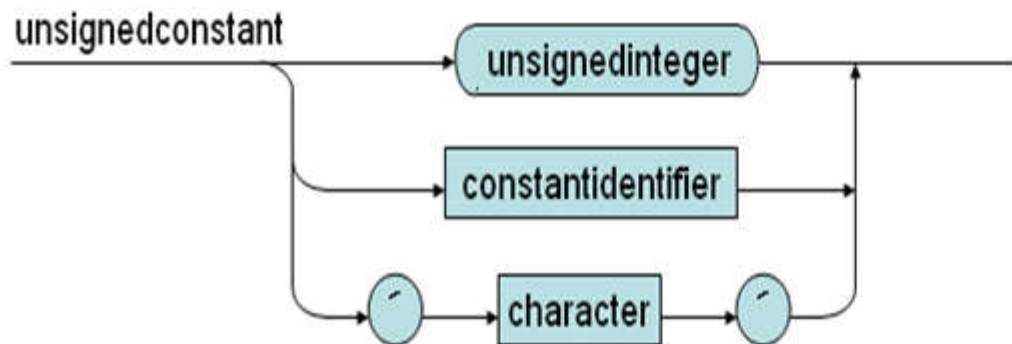
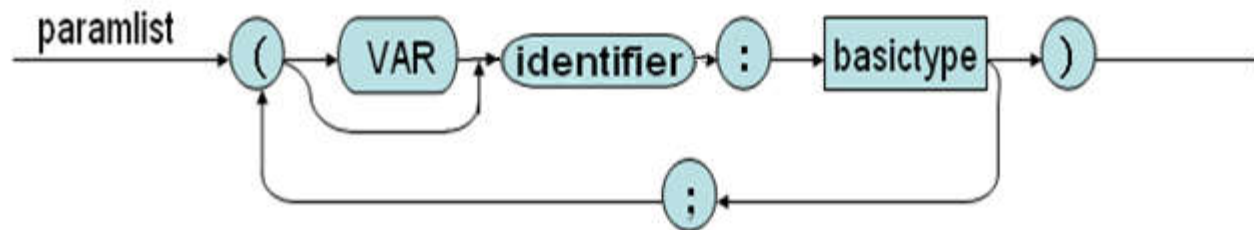


# Syntax Diagrams of KPL(block)

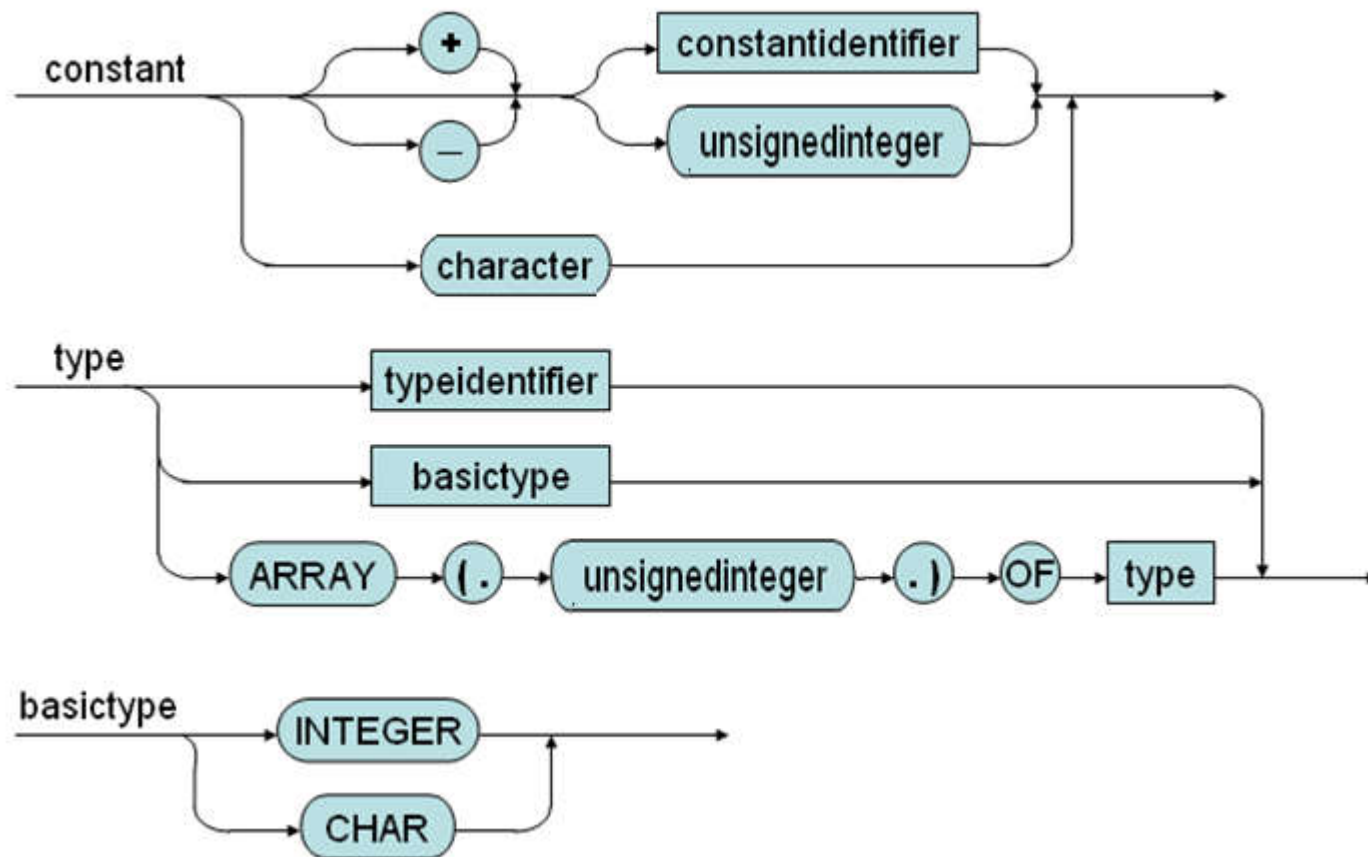


# Syntax Diagrams of KPL

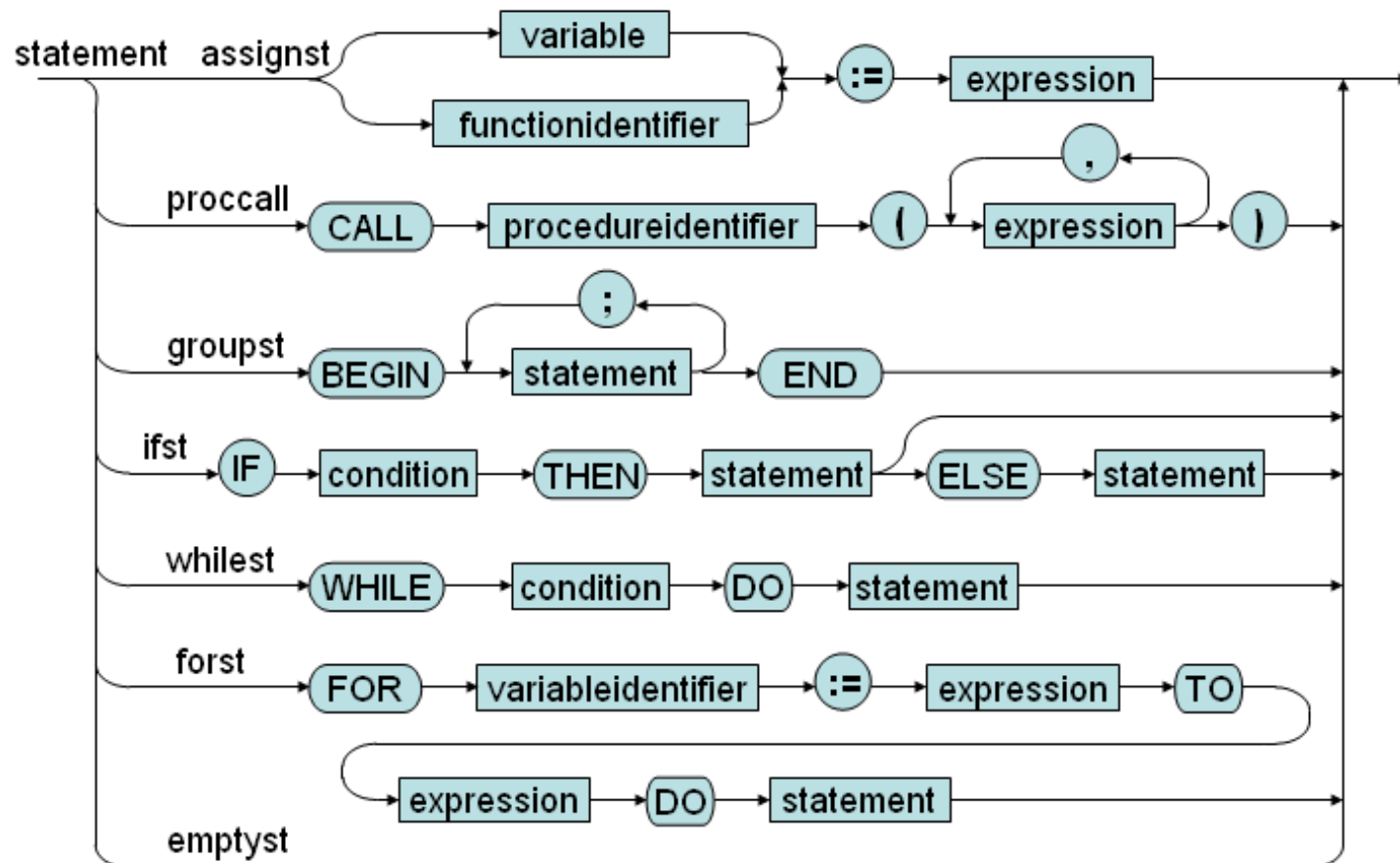
(list of parameters, unsigned constant)



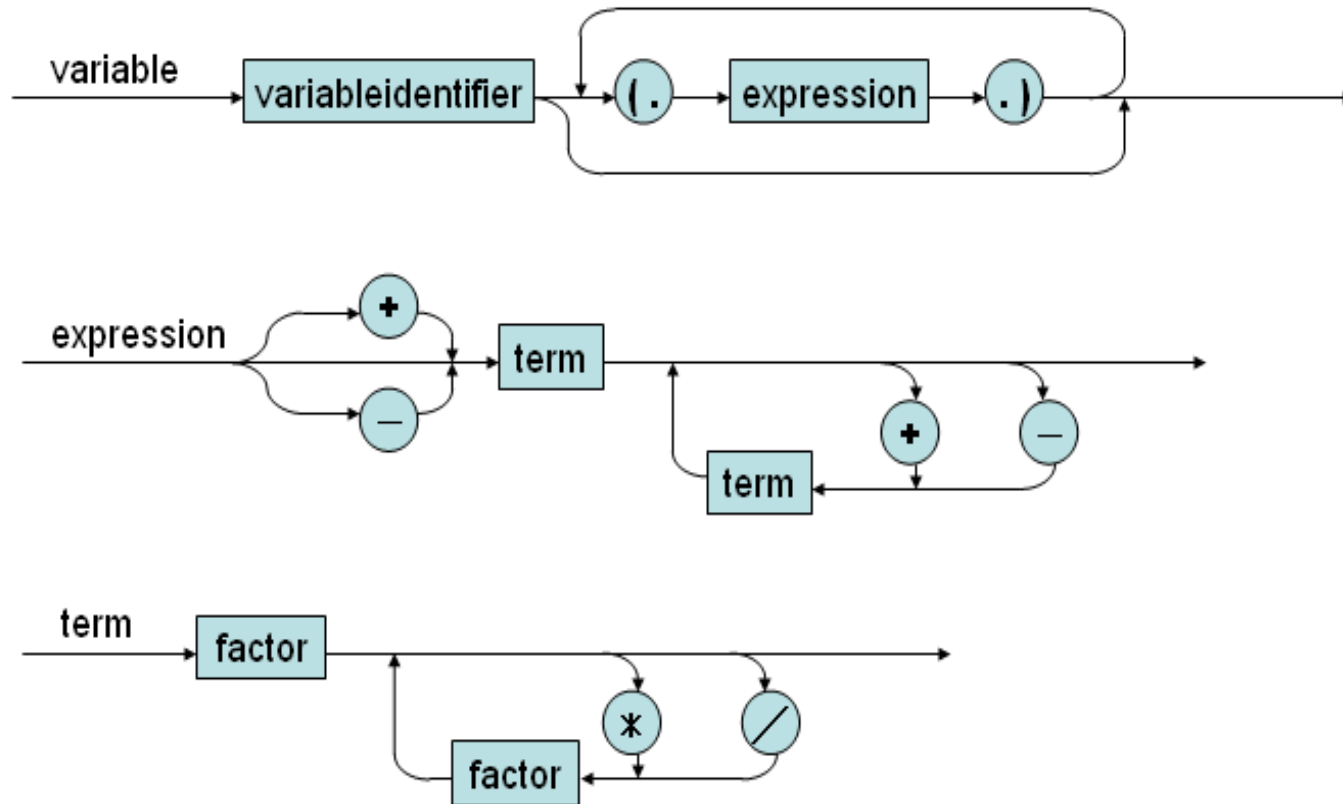
# Syntax Diagrams of KPL (declarations)



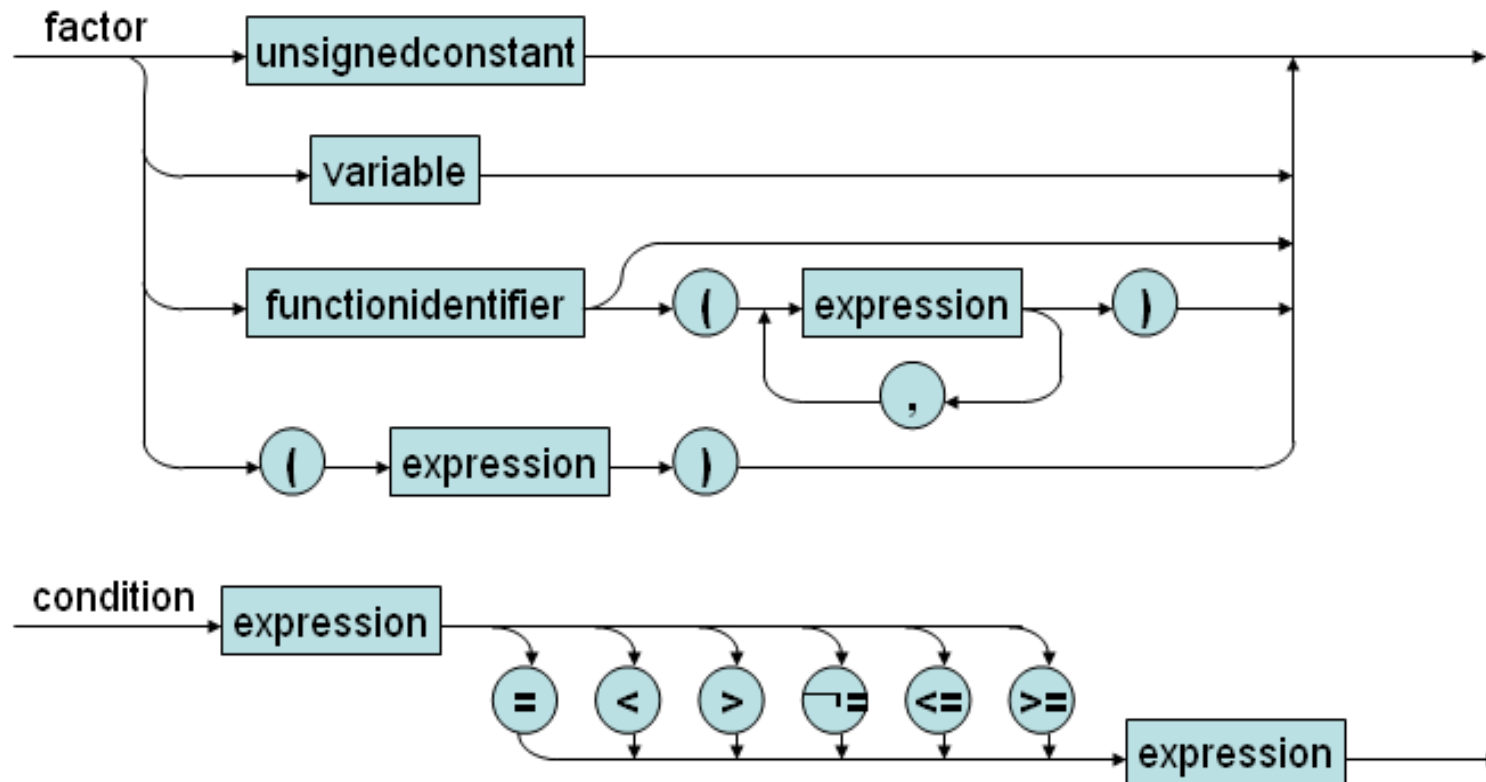
# Syntax Diagrams of KPL (statement)



# Syntax Diagrams of KPL (variable, expression, term)

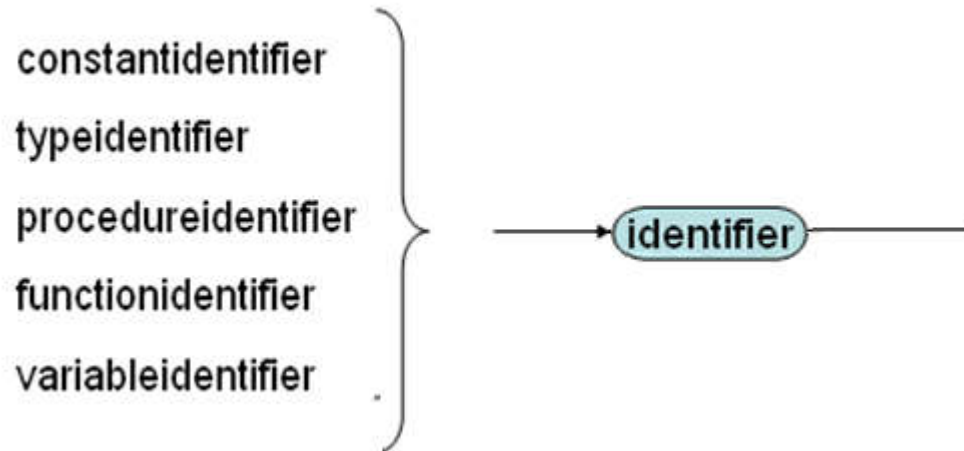


# Syntax Diagrams of KPL (factor, condition)





# Syntax Diagrams of KPL (identifier, unsigned integer)



# Exercise: a KPL program

Write a program that asks the user to type the value of an integer and compute its factorial.

# Solution 1

```
program example1; (* Factorial *)
var n : integer; i: integer; f:integer;
BEGIN

n := readi;

f:=1;
if n >=2
begin
for i:= 2 to n do
f:= f*i;
call writeln;
call writeI(f);
end;
END. (* Factorial *)
```

## Solution 2 (using KPL functions)

```
program example2; (* Factorial *)
var n : integer;
function f(k : integer) : integer;
begin
    If k = 0 Then f := 1 Else f := k * f (k
- 1) ;
end;

BEGIN
    n := readI;
    call writeln;
    call writeI(f(n)) ;
END. (* Factorial *)
```