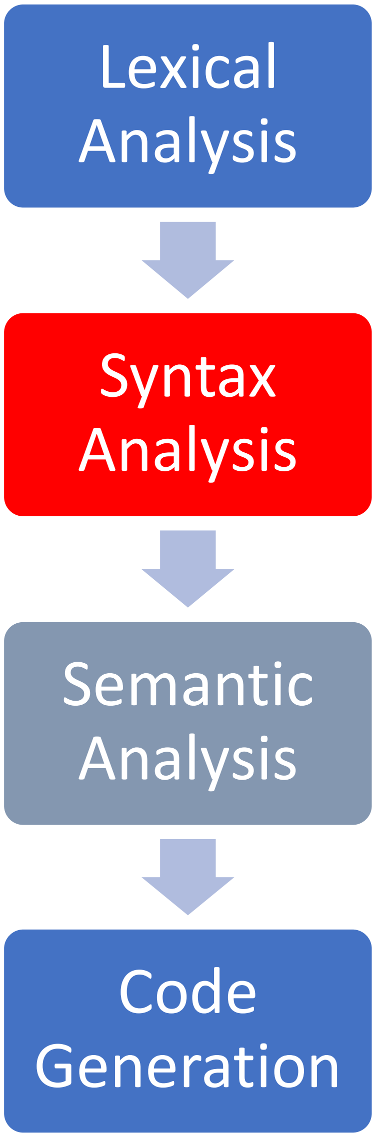
Experiment in Compiler Construction

Parser design

Content

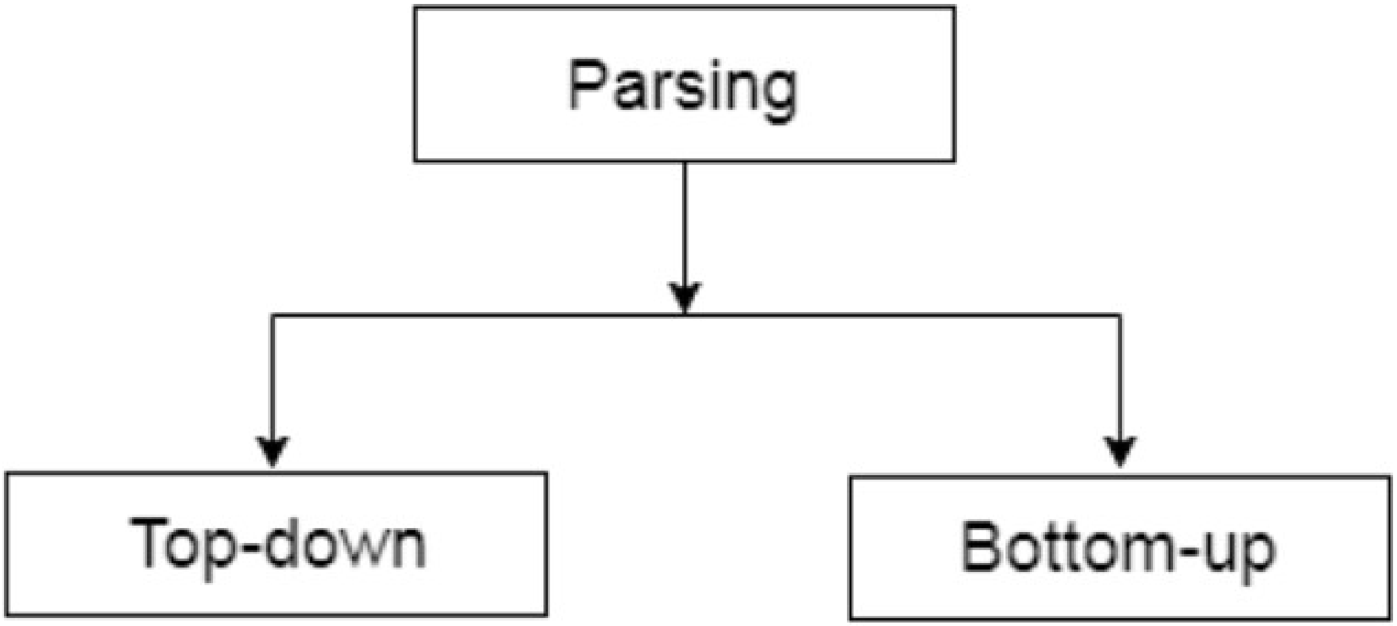
- Overview
- KPL grammar
- Parser implementation

Tasks of a parser



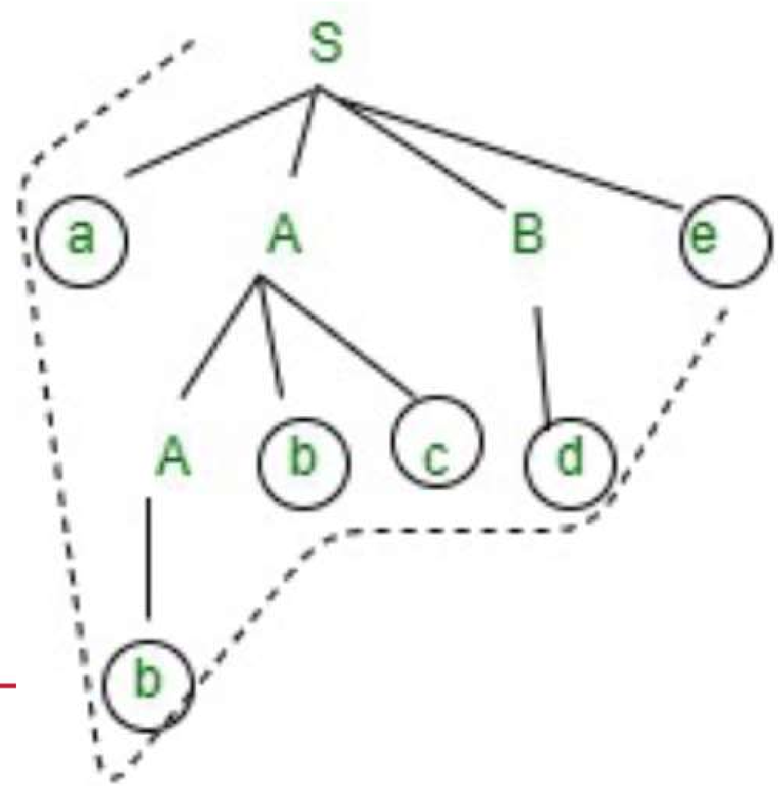
- Check the syntactic structure of a given program
  - Syntactic structure is given by Grammar
- Invoke semantic analysis and code generation
  - In an one-pass compiler, this module is very important since this forms the skeleton of the compiler

Classification of parsing techniques



Top down parsing

- Construct a parse tree from the root to the leaves, reading the given string from left-to-right
- It follows left most derivation.
- If a variable contains more than one possibilities, selecting 1 is difficult.
- Example: Given grammar G with a set of production rules
  - G: (1) $S \to a\ ABe$
    (2, 3) $A \to Abc|b$
    (4) $B \to d$
  - input: abbcde

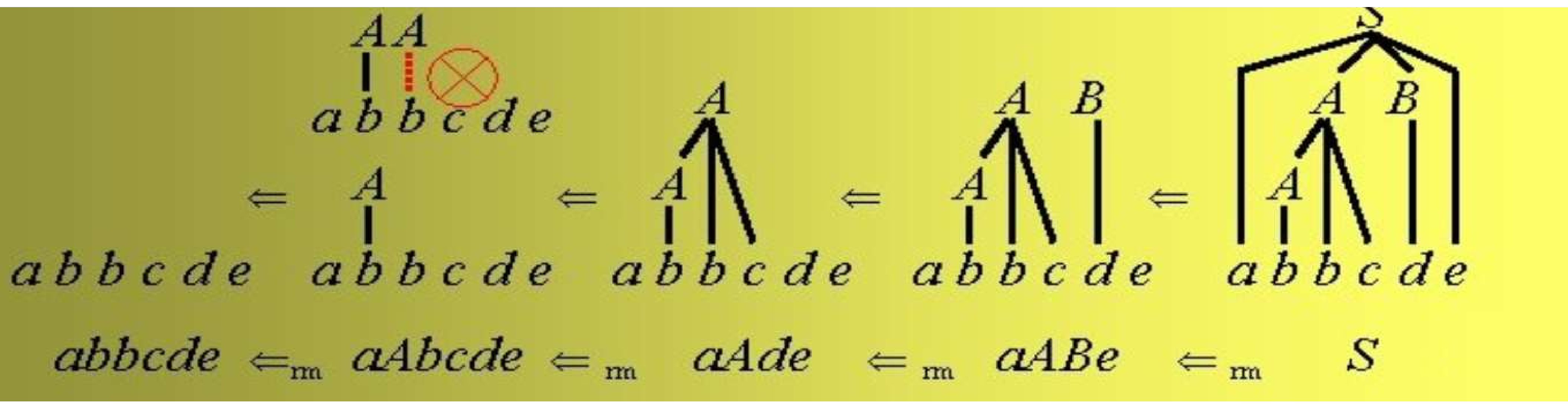


Bottom up parsing

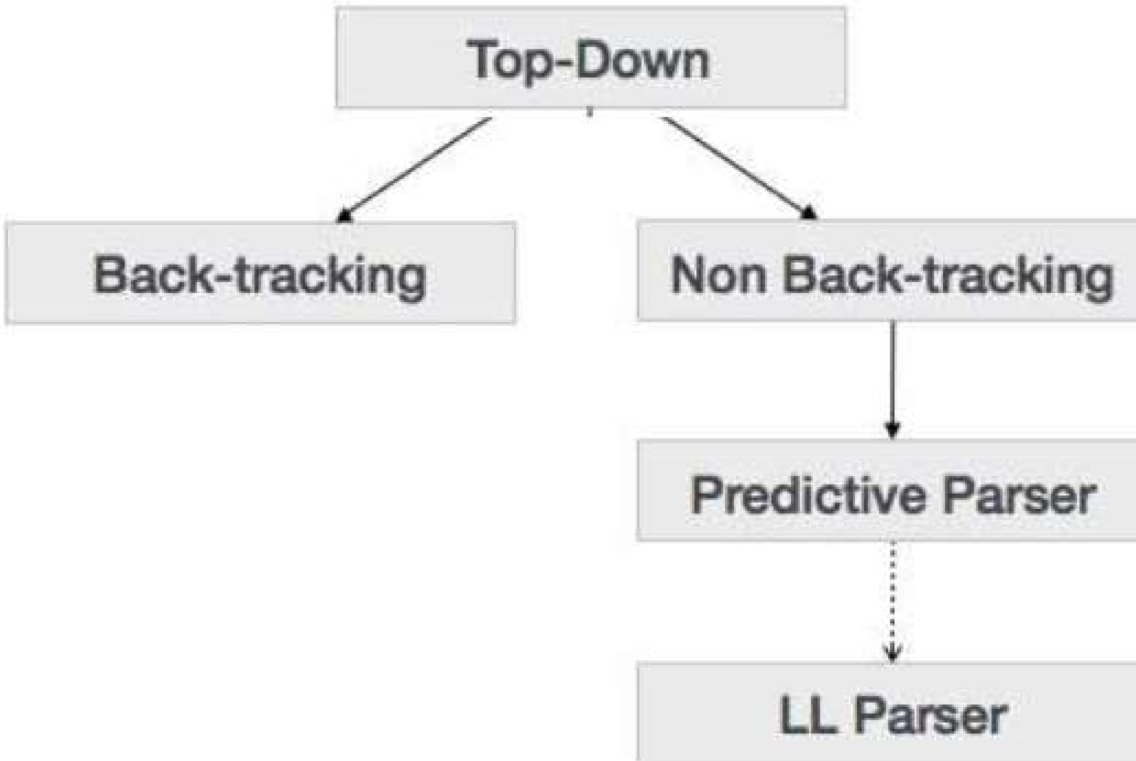
- Construct a parse tree from the leaves to the root: left-to-right reduction
- It follows the rightmost derivation
- Example: Given grammar G with a set of production rules
  - G: (1) $S \rightarrow a\ ABe$

    $A \rightarrow Abc|b$

    $B \rightarrow d$
  - input: abbcde



Top down parsing methods



Recursive-descent parsing

- A top-down parsing method
- *Descent*: the direction in which the parse tree is traversed (or built).
- Use a set of *mutually recursive* procedures (one procedure for each nonterminal symbol)
  - Start the parsing process by calling the procedure that corresponds to the start symbol
  - Each production becomes one branch in procedure for its LHS
- We consider a **special type of recursive-descent parsing** called **predictive parsing**
  - Use a lookahead symbol to decide which production to use

Recursive Descent Parsing

- For every BNF rule (production) of the form

  <phrase1> → E

  the parser defines a function to parse phrase1 whose body is to parse the rule E

  ```
  void compilePhrase1( )
  {   /* parse the rule E */   }
  ```

- Where E consists of a sequence of non-terminal and terminal symbols
- Requires no left recursion in the grammar.

Parsing a rule

- A sequence of non-terminal and terminal symbols,

  $Y_1\ Y_2\ Y_3 \ldots Y_n$

  is recognized by parsing each symbol in turn
- For each non-terminal symbol, Y, call the corresponding parse function

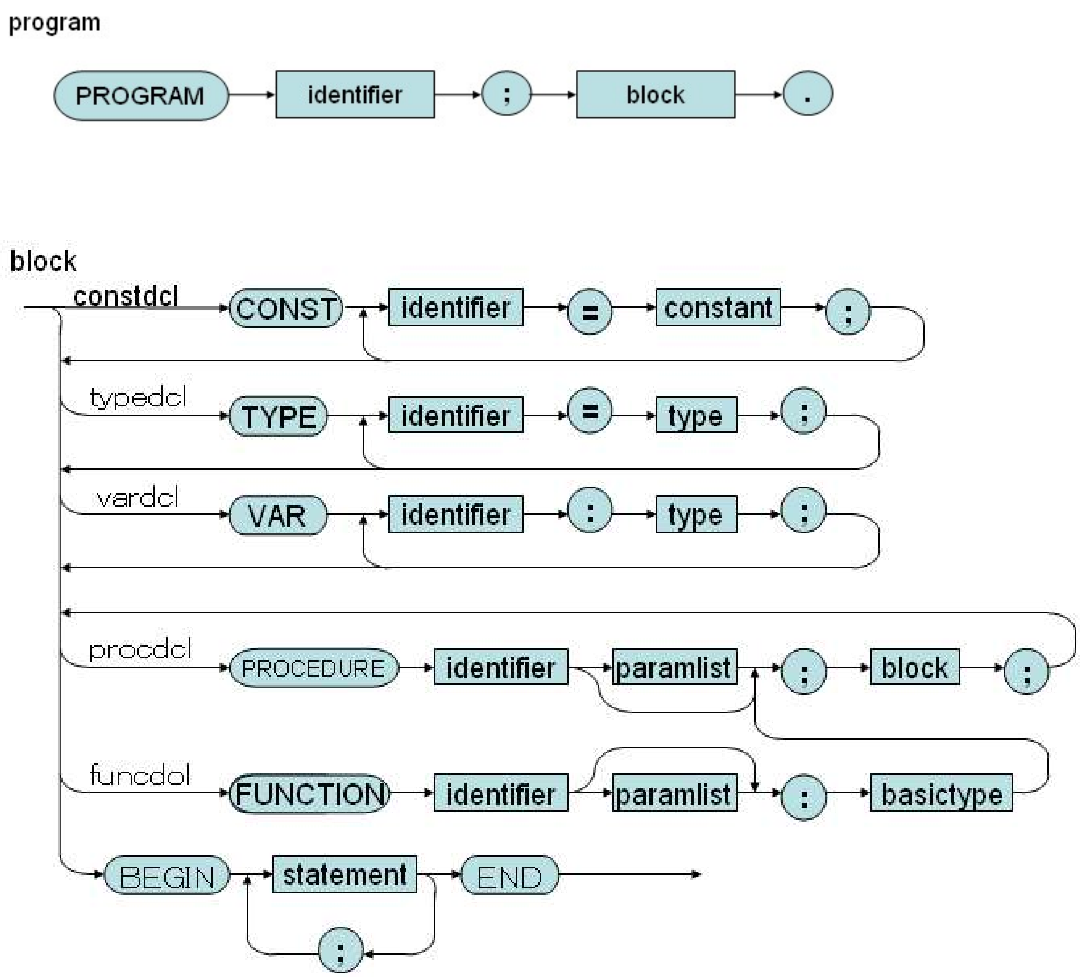
  `compileY`
- For each terminal symbol, y, call a function

  `eat(y)`

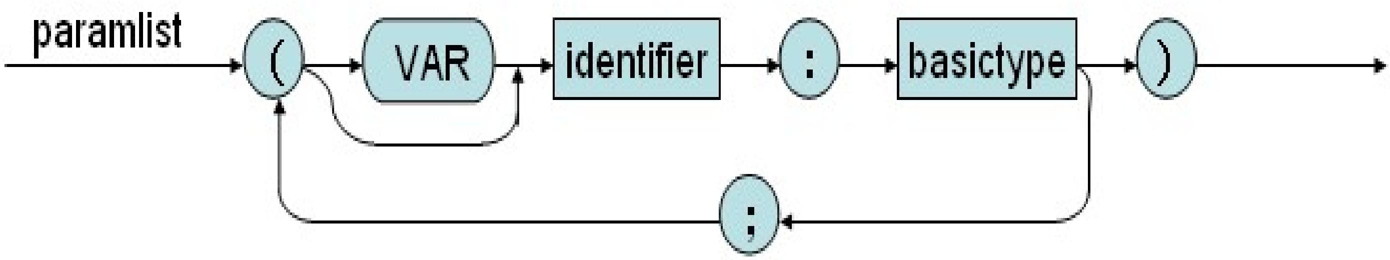
  that will check if y is the next symbol in the source program
  - The terminal symbols are the token types from the lexical analyzer
  - If the variable currentsymbol always contains the next token:

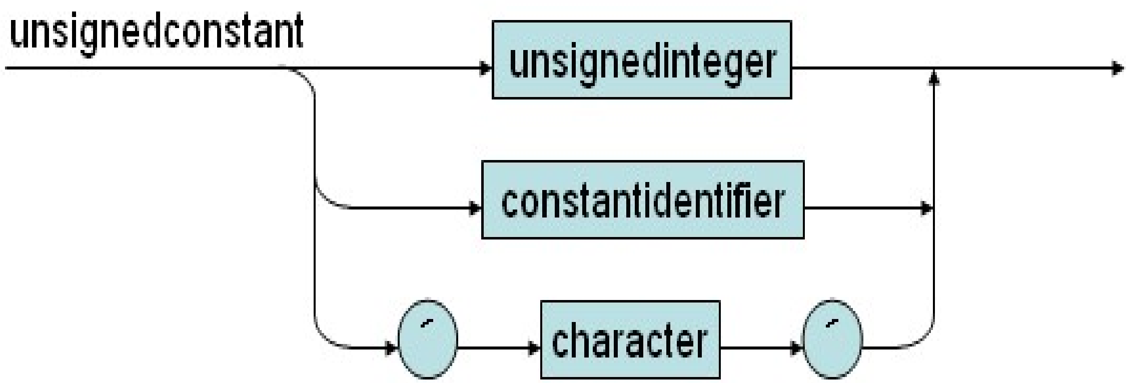
```
eat(y):
     if (currentsymbol == y)
     then getNextToken()
     else SyntaxError()
```

Syntax diagram of KPL

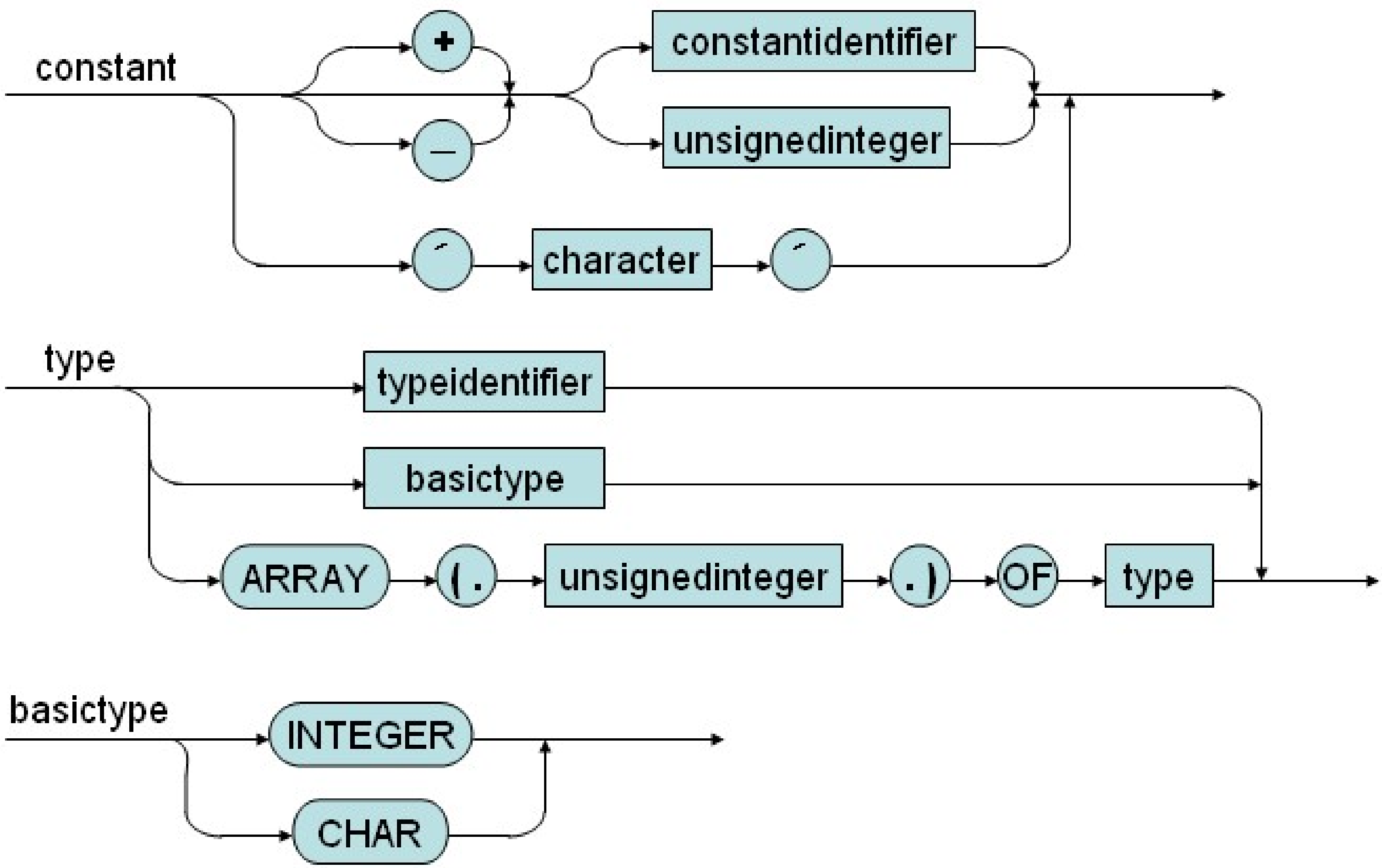


Syntax diagram of KPL

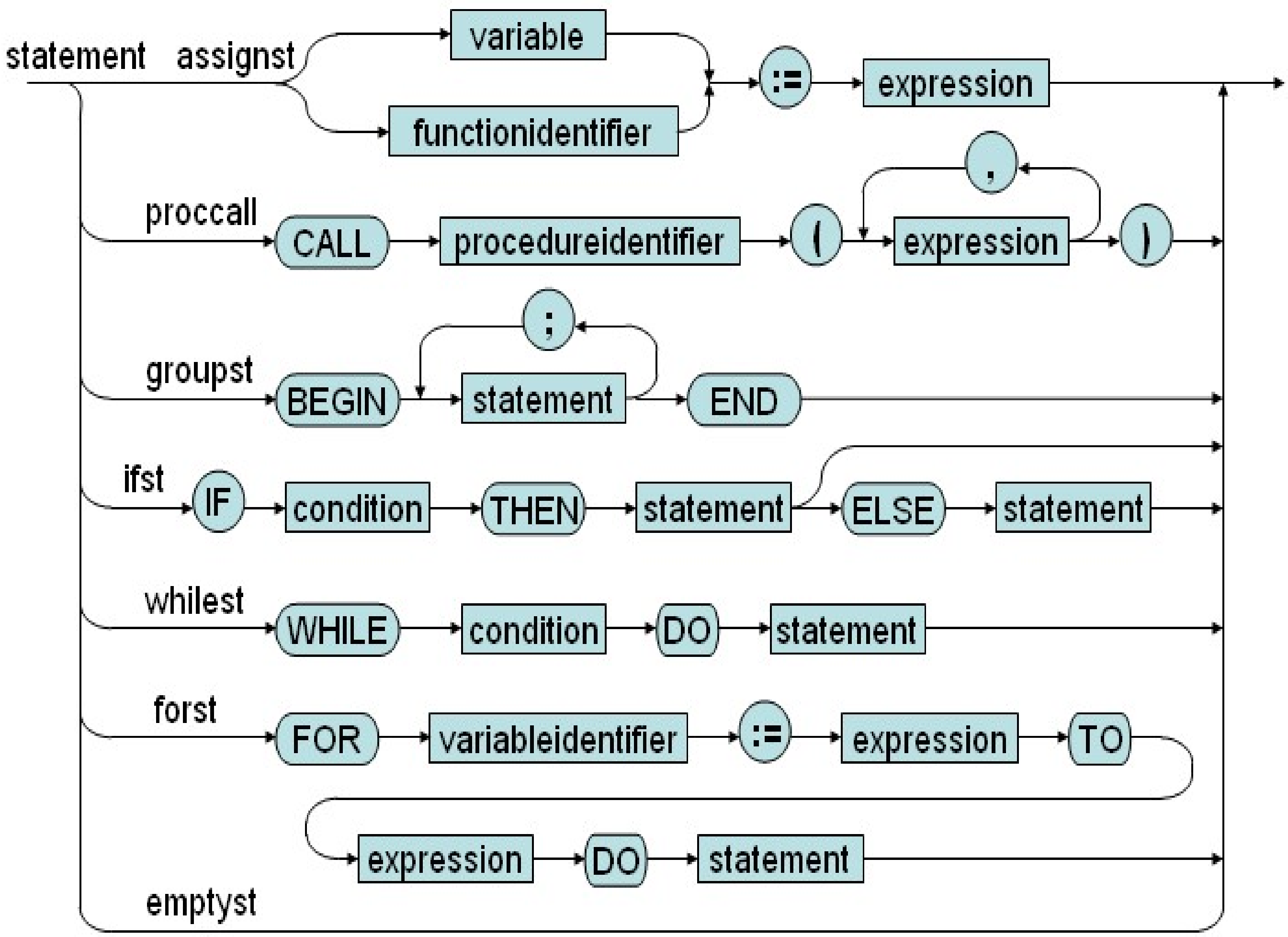




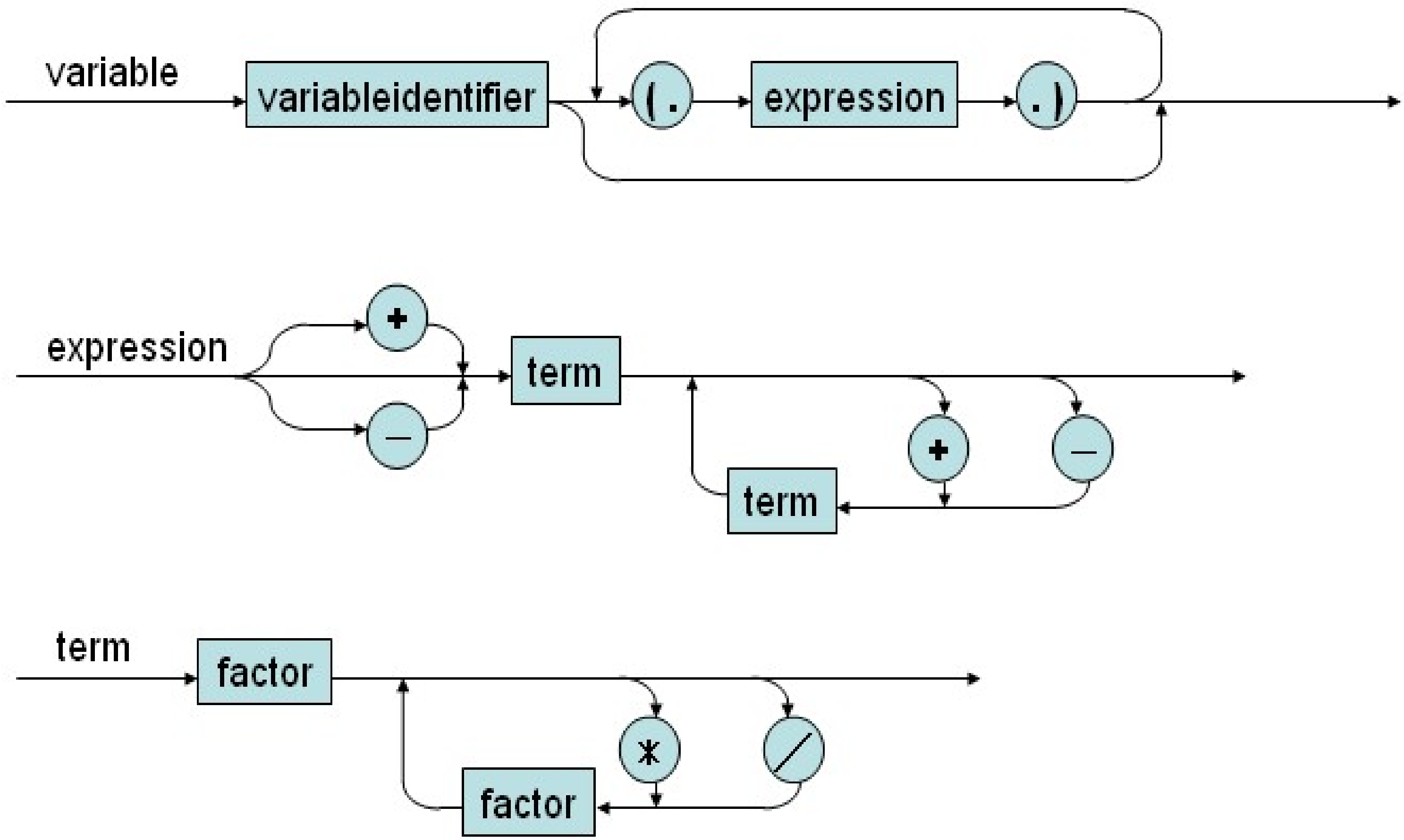
Syntax diagram of KPL



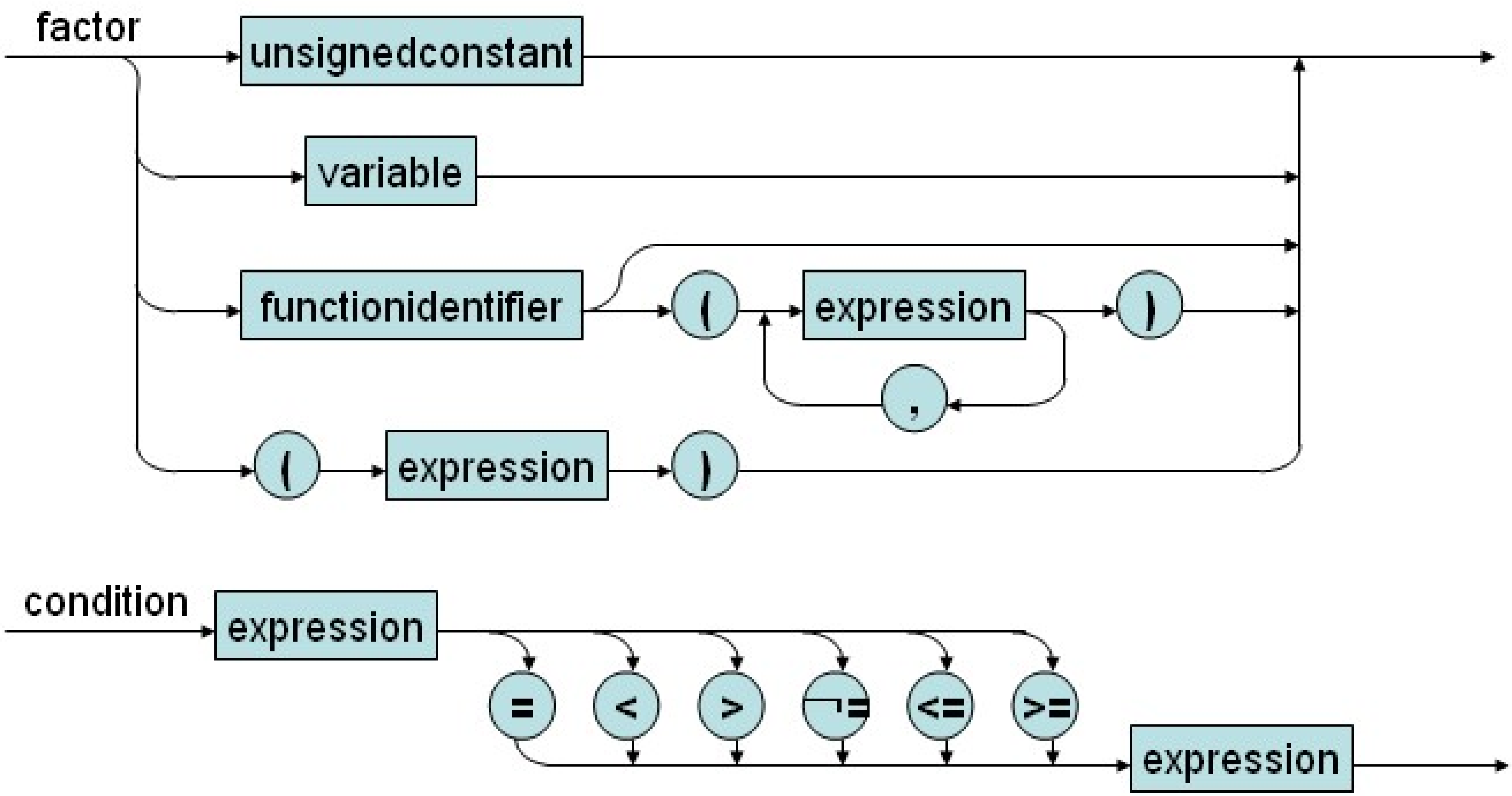
Syntax diagram of KPL



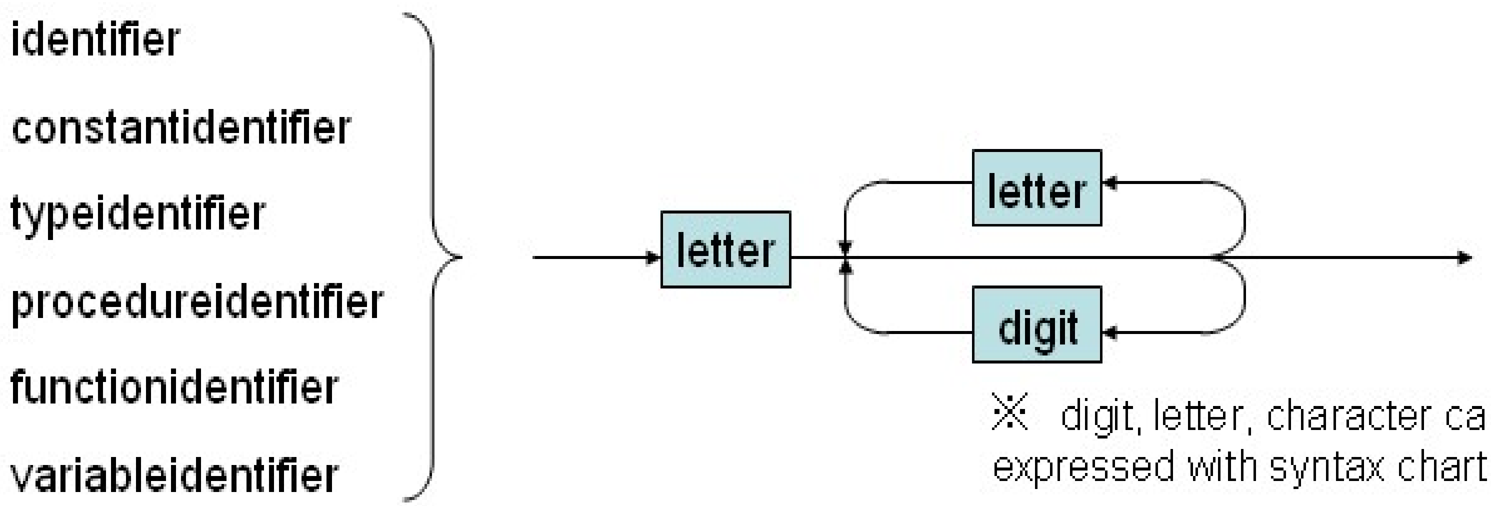
Syntax diagram of KPL



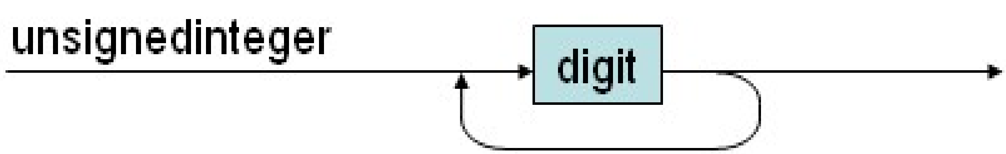
Syntax diagram of KPL



Syntax diagram of KPL



※ digit, letter, character can be expressed with syntax chart.



KPL Grammar in BNF

- Construct a grammar G based on syntax diagram
- Perform left recursive elimination (already)
- Perform left factoring

KPL Grammar in BNF

```
01) <Prog> ::= KW_PROGRAM TK_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) <Block4> ::= <Block5>

10) <Block5> ::= KW_BEGIN <Statements> KW_END
```

KPL Grammar in BNF

```
11) <ConstDecls>::= <ConstDecl> <ConstDecls>
12) <ConstDecls>::= ε

13) <ConstDecl> ::= TK_IDENT SB_EQUAL <Constant> SB_SEMICOLON

14) <TypeDecls> ::= <TypeDecl> <TypeDecls>
15) <TypeDecls> ::= ε

16) <TypeDecl>  ::= TK_IDENT SB_EQUAL <Type> SB_SEMICOLON

17) <VarDecls>::= <VarDecl> <VarDecls>
18) <VarDecls>::= ε
19) <VarDecl>   ::= TK_IDENT SB_COLON <Type> SB_SEMICOLON

20) <SubDecls> ::= <FunDecl> <SubDecls>
21) <SubDecls> ::= <ProcDecl> <SubDecls>
22) <SubDecls> ::= ε
```

KPL Grammar in BNF

```
23) <FunDecl>   ::= KW_FUNCTION TK_IDENT <Params> SB_COLON
                    <BasicType> SB_SEMICOLON <Block> SB_SEMICOLON

24) <ProcDecl>  ::= KW_PROCEDURE TK_IDENT <Params> SB_SEMICOLON
                    <Block> SB_SEMICOLON

25) <Params>    ::= SB_LPAR <Param> <Params2> SB_RPAR
26) <Params>    ::= ε

27) <Params2> ::= SB_SEMICOLON <Param> <Params2>
28) <Params2> ::= ε

29) <Param>   ::= TK_IDENT SB_COLON <BasicType>
30) <Param>   ::= KW_VAR TK_IDENT SB_COLON <BasicType>
```

KPL Grammar in BNF

```
31) <Type> ::= KW_INTEGER
32) <Type> ::= KW_CHAR
33) <Type> ::= TK_IDENT
34) <Type> ::= KW_ARRAY SB_LSEL TK_NUMBER SB_RSEL KW_OF <Type>

35) <BasicType> ::= KW_INTEGER
36) <BasicType> ::= KW_CHAR

37) <UnsignedConstant> ::= TK_NUMBER
38) <UnsignedConstant> ::= TK_IDENT
39) <UnsignedConstant> ::= TK_CHAR

40) <Constant> ::= SB_PLUS <Constant2>
41) <Constant> ::= SB_MINUS <Constant2>
42) <Constant> ::= <Constant2>
43) <Constant> ::= TK_CHAR

44) <Constant2>::= TK_IDENT
45) <Constant2>::= TK_NUMBER
```

KPL Grammar in BNF

```
46) <Statements> ::= <Statement> <Statements2>

47) <Statements2> ::= SB_SEMICOLON <Statement> <Statements2>
48) <Statements2> ::= ε

49) <Statement> ::= <AssignSt>
50) <Statement> ::= <CallSt>
51) <Statement> ::= <GroupSt>
52) <Statement> ::= <IfSt>
53) <Statement> ::= <WhileSt>
54) <Statement> ::= <ForSt>
55) <Statement> ::= ε
```

KPL Grammar in BNF

```
56) <AssignSt> ::= <Variable> SB_ASSIGN <Expression>
57) <AssignSt> ::= TK_IDENT SB_ASSIGN <Expression>

58) <CallSt>   ::= KW_CALL TK_IDENT <Arguments>

59) <GroupSt>  ::= KW_BEGIN <Statements> KW_END

60) <IfSt>     ::= KW_IF <Condition> KW_THEN <Statement> <ElseSt>

61) <ElseSt>   ::= KW_ELSE <Statement>
62) <ElseSt>   ::= ε

63) <WhileSt>  ::= KW_WHILE <Condition> KW_DO <Statement>
64) <ForSt>    ::= KW_FOR TK_IDENT SB_ASSIGN <Expression> KW_TO
                 <Expression> KW_DO <Statement>
```

KPL Grammar in BNF

```
65) <Arguments> ::= SB_LPAR <Expression> <Arguments2> SB_RPAR
66) <Arguments> ::= ε

67) <Arguments2>::= SB_COMMA <Expression> <Arguments2>
68) <Arguments2>::= ε

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> ::= ε

82) <Term> ::= <Factor> <Term2>

83) <Term2> ::= SB_TIMES <Factor> <Term2>
84) <Term2> ::= SB_SLASH <Factor> <Term2>
85) <Term2> ::= ε
86) <Factor> ::= <UnsignedConstant>
87) <Factor> ::= <Variable>
88) <Factor> ::= <FunctionApptication>
89) <Factor> ::= SB_LPAR <Expression> SB_RPAR
```

KPL Grammar in BNF

```
90) <Variable> ::= TK_IDENT <Indexes>

91) <FunctionApplication> ::= TK_IDENT <Arguments>

92) <Indexes> ::= SB_LSEL <Expression> SB_RSEL <Indexes>
93) <Indexes> ::= ε
```

Input – output in KPL

- Input: Use functions
  - ReadI: Read an integer. No parameter
  - ReadC: Read a character. No parameter

  Example

  var a: integer;

  a:= ReadI;
- Output: Use procedures
  - WriteI: Print an integer. 1 parameter
  - WriteC: Print a character. 1 parameter
  - WriteLn: Print the newline character.

  Example

  call WriteI(a);

  call WriteLn;

KPL program

- Write a function that calculates the square of an integer
- Write a program to calculate the sum of the squares of the first n natural numbers. n is read from the keyboard

Solution

```
program example5;
 (* sum of the squares of the first n natural
numbers *)
var n : integer;i: integer;sum: integer;

function f(k : integer) : integer;
   begin
    f := k * k;
   end;

BEGIN
     n := readI;
     sum :=0;
     for i:=1 to n do
          sum:= sum + f(i);
     call writeln;
     call writeI(sum);
END. (* example5*)
```

Parser implemetation

- In general, KPL is a LL(1) grammar
- design a top-down parser
  - *lookAhead* token
  - Parsing terminals
  - Parsing non-terminals
    - Constructing a parsing table
      - Computing FIRST() and FOLLOW()
- Example

```
02) Block   ::= KW_CONST ConstDecl ConstDecls Block2
 =>RHS1
03) Block   ::= Block2
 =>RHS2
FIRST(RHS1)={KW_CONST}
FIRST(RHS2)={KW_TYPE, KW_VAR, KW_FUNCTION,
 KW_PROCEDURE,KW_BEGIN}
FIRST(RHS1)∩ FIRST(RHS2)=∅

LookAhead =KW_BEGIN =>RHS2 is chosen =>LL(1)
```

lookAhead token

- Look ahead the next token

```
Token *currentToken;
Token *lookAhead;

void scan(void) {
  Token* tmp = currentToken;
  currentToken = lookAhead;
  lookAhead = getValidToken();
  free(tmp);
}
```

Parsing terminal symbol

```
void eat(TokenType tokenType) {
  if (lookAhead->tokenType == tokenType) {
    printToken(lookAhead);
    scan();
  } else
   missingToken(tokenType, lookAhead->lineNo, lookAhead-
   >colNo);
}
```

Invoking the parser

```
int compile(char *fileName) {
  if (openInputStream(fileName) == IO_ERROR)
    return IO_ERROR;

  currentToken = NULL;
  lookAhead = getValidToken();

  compileProgram();

  free(currentToken);
  free(lookAhead);
  closeInputStream();
  return IO_SUCCESS;
}
```

Parsing non-terminal symbol

Example: **Program**

```
1)<Prog>::= KW_PROGRAM TK_IDENT SB_SEMICOLON <Block> SB_PERIOD

void compileProgram(void) {
  assert("Parsing a Program ....");
  eat(KW_PROGRAM);
  eat(TK_IDENT);
  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_PERIOD);
  assert("Program parsed!");
}
```

Parsing statements

```
Example: Statement
FIRST(<Statement>) = {TK_IDENT, KW_CALL, KW_BEGIN, KW_IF, KW_WHILE,
                      KW_FOR, ε}
FOLLOW(<Statement>) = {SB_SEMICOLON, KW_END, KW_ELSE}

/* Predict parse table for Expression */
Input                 Production
----------------------------------------------------
TK_IDENT              49) <Statement> ::= <AssignSt>
KW_CALL               50) <Statement> ::= <CallSt>
KW_BEGIN              51) <Statement> ::= <GroupSt>
KW_IF                 52) <Statement> ::= <IfSt>
KW_WHILE              53) <Statement> ::= <Whilst?
KW_FOR                54) <Statement> ::= <ForSt?
----------------------------------------------------
SB_SEMICOLON          55) ε
KW_END                55) ε
KW_ELSE               55) ε
----------------------------------------------------
Others                Error
```

Parsing a statement

Example: Statement

```
void compileStatement(void)
{
  switch (lookAhead-
>tokenType) {
  case TK_IDENT:
    compileAssignSt();
    break;
  case KW_CALL:
    compileCallSt();
    break;
  case KW_BEGIN:
    compileGroupSt();
    break;
  case KW_IF:
    compileIfSt();
    break;
  case KW_WHILE:
    compileWhileSt();
    break;
 case KW_FOR:
    compileForSt();
    break;
    // check FOLLOW tokens
  case SB_SEMICOLON:
  case KW_END:
  case KW_ELSE:
    break;
    // Error occurs
  default:

error(ERR_INVALIDSTATEMENT,
lookAhead->lineNo,
lookAhead->colNo);
    break;
  }
}
```

LHS with more than 1 RHS

Two alternatives for Basic Type

35) <BasicType> ::= KW_INTEGER

36) <BasicType> ::= KW_CHAR

```
void compileBasicType(void) {
  switch (lookAhead->tokenType) {
  case KW_INTEGER:
    eat(KW_INTEGER);
    break;
  case KW_CHAR:
    eat(KW_CHAR);
    break;
  default:
    error(ERR_INVALIDBASICTYPE, lookAhead->lineNo,
lookAhead->colNo);
    break;
  }
}
```

Loop processing

Loop for sequence of constant declarations: Recursion OK, you should process the FOLLOW SET

10) <ConstDecls>::= <ConstDecl> <ConstDecls>

11) <ConstDecls>::= ε

```
void compileConstDecls(void) {
  while (lookAhead->tokenType == TK_IDENT)
    compileConstDecl();
}
```

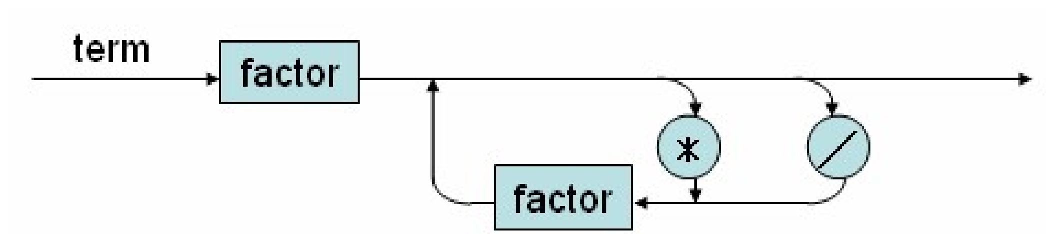
Sometimes you should refer to syntax diagrams

Syntax of Term (using BNF)

```
82) <Term> ::= <Factor> <Term2>

83) <Term2> ::= SB_TIMES <Factor> <Term2>
84) <Term2> ::= SB_SLASH <Factor> <Term2>
85) <Term2> ::= ε
```

Syntax of Term (using Syntax Diagram)

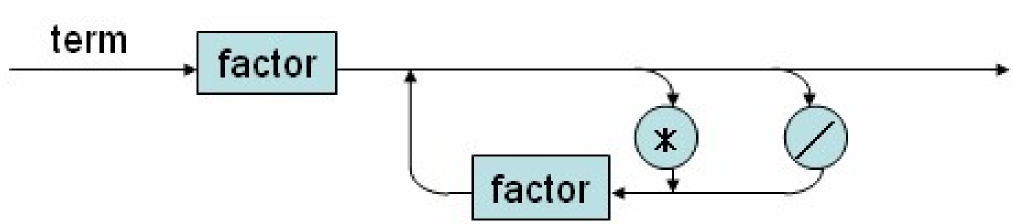


Process rules for Term : 2 functions with Follow set checking

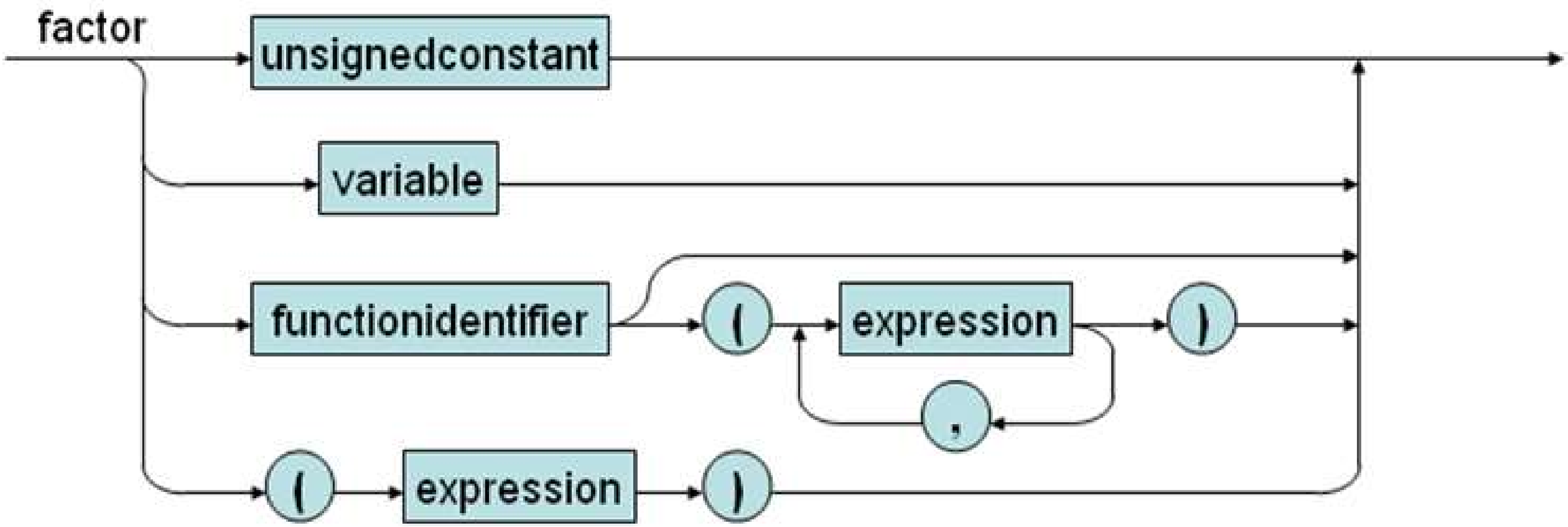
```
void compileTerm(void)
{ compileFactor();
  compileTerm2();
}
void compileTerm2(void) {
  switch (lookAhead->tokenType) {
  case SB_TIMES:
    eat(SB_TIMES);
    compileFactor();
    compileTerm2();
    break;
  case SB_SLASH:
    eat(SB_SLASH);
    compileFactor();
    compileTerm2();
    break;
// check the FOLLOW set
  case SB_PLUS:
  case SB_MINUS:
  case KW_TO:
  case KW_DO:
case SB_RPAR:
  case SB_COMMA:
  case SB_EQ:
  case SB_NEQ:
  case SB_LE:
  case SB_LT:
  case SB_GE:
  case SB_GT:
  case SB_RSEL:
  case SB_SEMICOLON:
  case KW_END:
  case KW_ELSE:
  case KW_THEN:
    break;
  default:
    error(ERR_INVALIDTERM,
lookAhead->lineNo, lookAhead-
>colNo);
  }
}
```

Process term with syntax diagram

```
void compileTerm(void)
{compileFactor();
  while(lookAhead->tokenType== SB_TIMES || lookAhead->tokenType ==
  SB_SLASH)
{switch (lookAhead->tokenType)
{
   case SB_TIMES:
      eat(SB_TIMES);
      compileFactor();
      break;
   case SB_SLASH:
      eat(SB_SLASH);
      compileFactor();
      break;
}
}
}
```



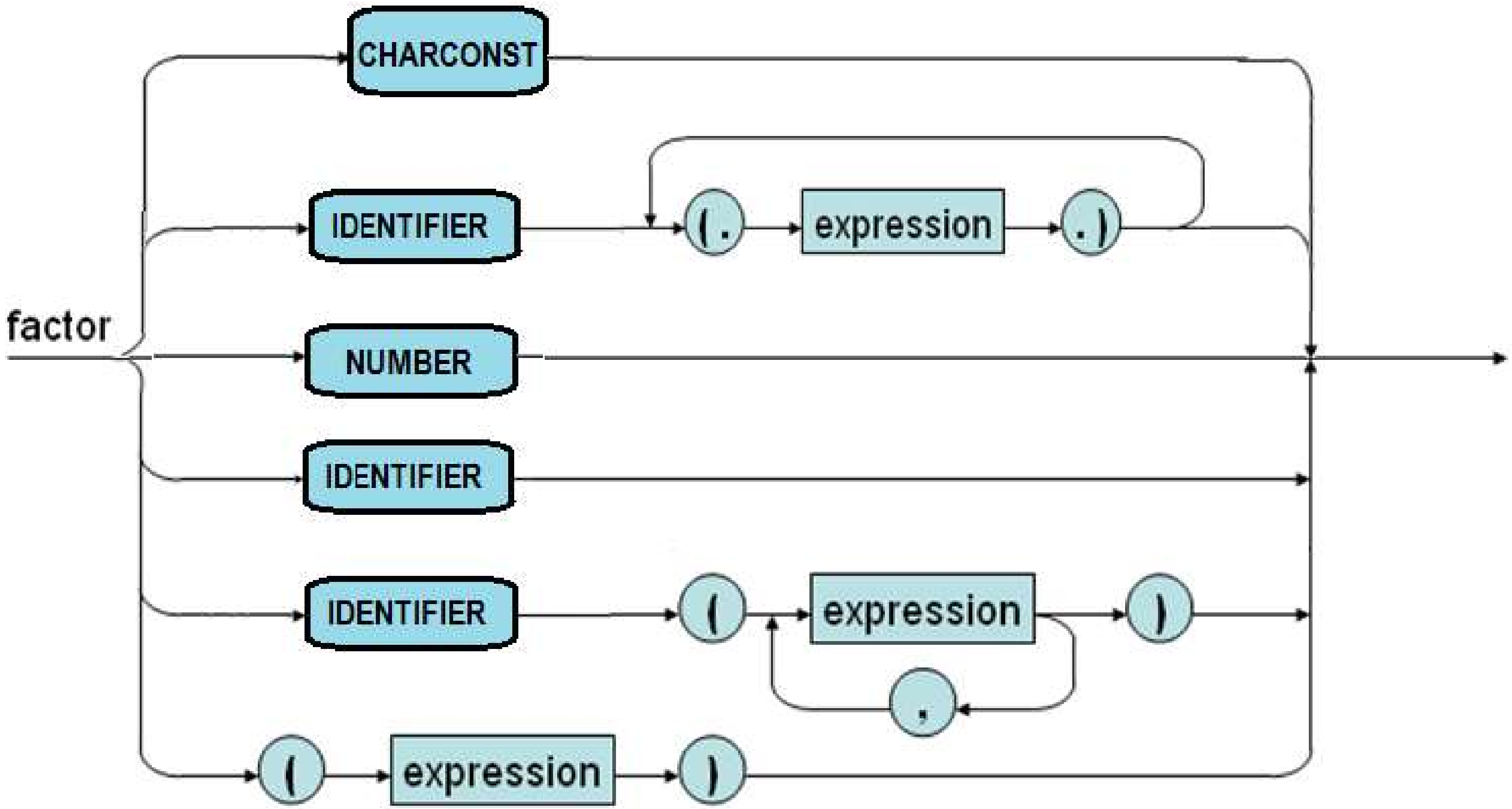
Syntax diagram of factor in KPL



FIRST(unsignedconstant) = {TK_NUMBER, TK_IDENT, TK_CHAR)
FIRST(variable) = {TK_IDENT}
FIRST(functioncall) = {TK_IDENT}
FIRST(unsignedconstant) ∩ FIRST(functioncall) = {TK_IDENT)
FIRST(variable) ∩ FIRST(functioncall) = {TK_IDENT}
FIRST(variable) ∩ FIRST(unsignedconstant)}= {TK_IDENT}

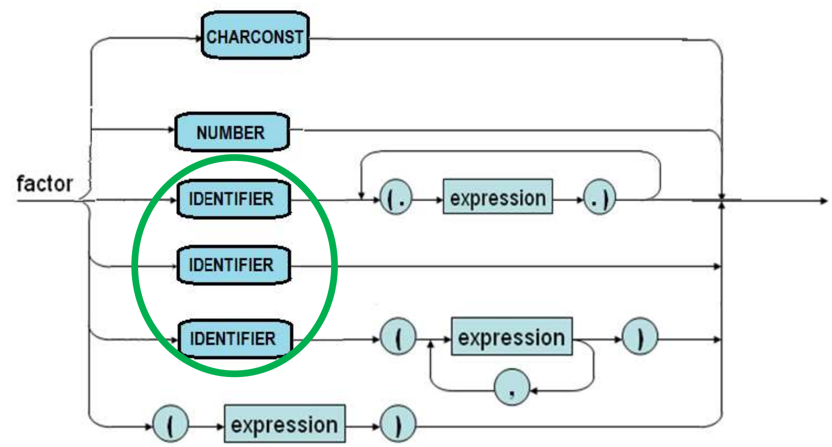
=>violation of LL(1) condition

After separating and merging



Compile a factor

```
void compileFactor(void) {
  switch (lookAhead->tokenType) {
  case TK_NUMBER:
    eat(TK_NUMBER);
    break;
  case TK_CHAR:
    eat(TK_CHAR);
    break;
  case TK_IDENT:
    eat(TK_IDENT);
    switch (lookAhead->tokenType) {
    case SB_LSEL:
      compileIndexes();
      break;
    case SB_LPAR:
      compileArguments();
      break;
    default: break;
    }
    break;
```




```
case SB_LPAR:
    eat(SB_LPAR);
    compileExpression();
    eat(SB_RPAR);
    break;
  default:
    error(ERR_INVALIDFACTOR,
lookAhead->lineNo, lookAhead->colNo);
  }
}
```