Unit 9.
Recursive descent parsing

Characteristics

- Used to parse LL(1) language
- Can be extended for parsing LL(k) grammars, but algorithms are complicated
- Parsing non LL(k) grammars can cause infinite loops

Recursive-descent parsing

- A top-down parsing method
- The term *descent* refers to 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 clause in procedure
- 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 parsePhrase1( )
{   /* 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 `parseY`
- For each terminal symbol, y, call a function
  `expect(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:

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

Simple parse function example

- Suppose that there was a grammar rule
  <program> →
  ‘**class’** <classname> ‘**{‘** <field-decl> <method-decl> ‘**}’**
- Then:
  - `parseProgram():`

```
expect(‘class’);
parseClassname();
expect(‘{‘);
parseFieldDecl();
parseMethodDecl();
expect(‘}’);
```

Look-Ahead

- In general, one non-terminal may have more than one production, so more than one function should be written to parse that non-terminal.
- Instead, we insist that we can decide which rule to parse just by looking ahead one symbol in the input

```
  <sentence> -> 'if' '(' <expr> ')' <block>
                      | 'while' '(' <expr> ')'
<block>
                                  ...
```

- Then parseSentence can have the form

```
    if (currentsymbol == "if")
          ... // parse first rule
    elsif (currentsymbol == "while")
          ...  // parse second rule
    ...
```

KPL Parser

- void error (const char msg[]);
- int accept(symbol s); // if the current symbol is s then get the next symbol
- int expect(symbol s); // check if s is the expected symbol?
- void factor(void);//compile factor
- void term(void);//compile term
- void expression(void); // compile expression
- void condition(void); // compile condition
- void statement(void); // compile statement
- void block(void); // compile a block
- void basictype(void); // compile baic types
- void program();// compile the whole program

Accept function

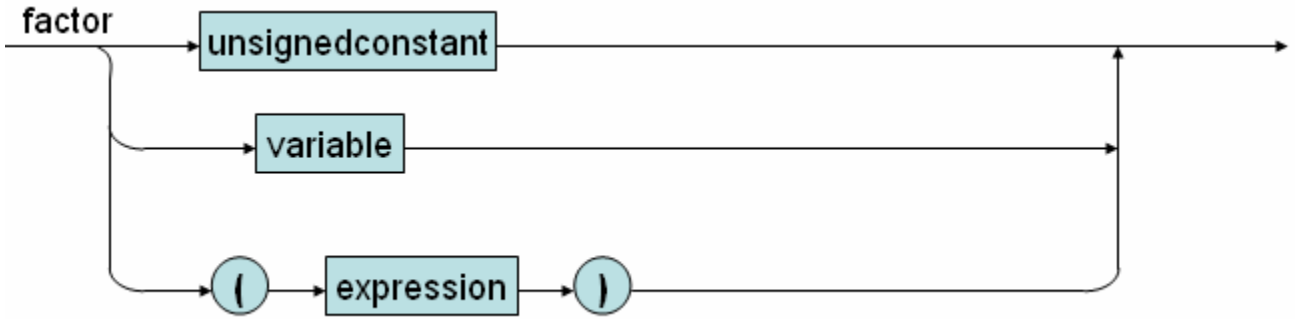
```
int accept(symbol s)
{
    if (sym == s)
    {
        getsym();
        return 1;
    }
    return 0;
}
```

Expect function

```
int expect(symbol s)
{
   if(accept(s))
        return 1;
   error("expect: unexpected symbol");
   return 0;
}
```

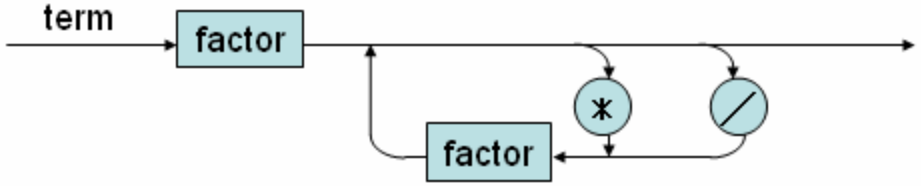
Compile factor function

```
void factor(void)
{if(accept(ident){}
else
    if(accept(number)) {}
        else if(accept(lparen))
    {
        expression();
        expect(rparen);
    }
    else
    {
        error("factor: syntax error");
        getsym();
    }
}
```



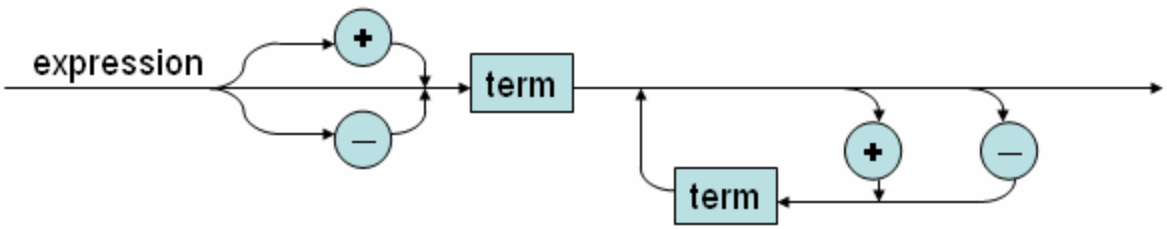
Compile term function

```
void term(void)
{
    factor();
    while(sym == times || sym == slash)
    {
        getsym();
        factor();
    }
}
```



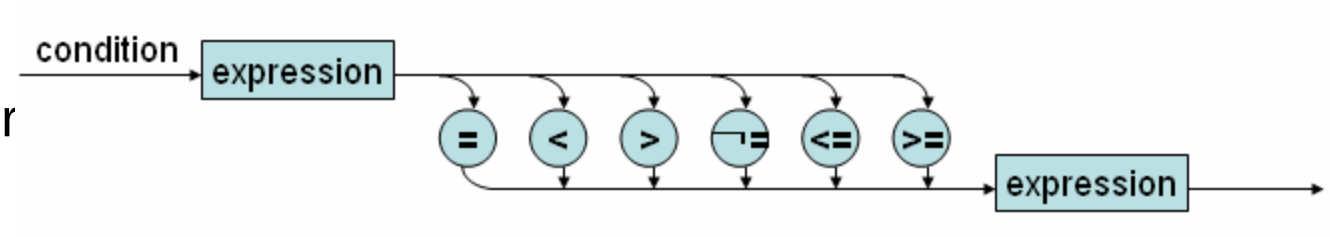
Compile expression function

```
void expression(void)
{
        if(sym == plus || sym == minus)
            getsym();
        term();
        while(sym == plus || sym == minus)
        {
            getsy
            term(
        }
}
```

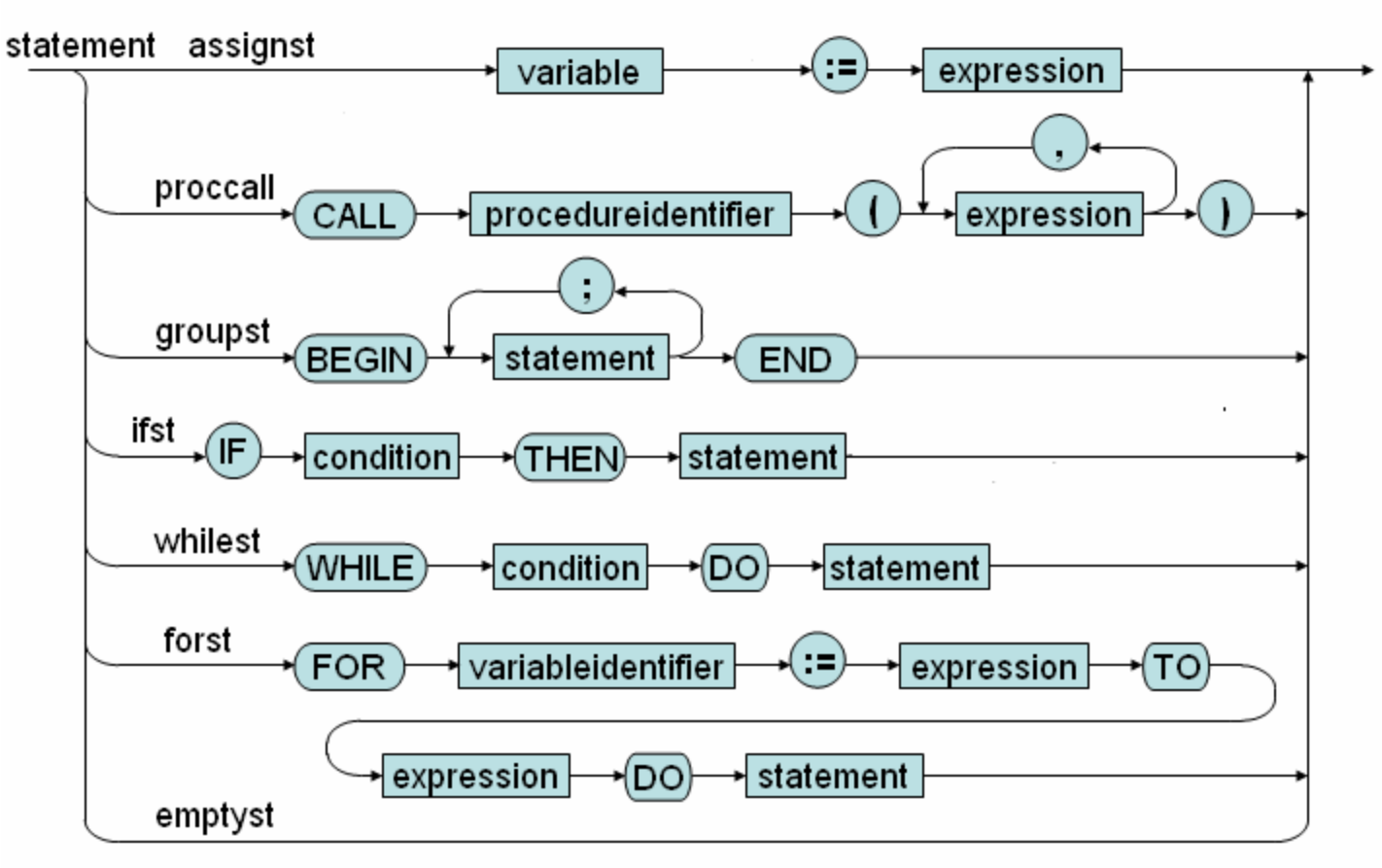


Compile condition function

```
void condition(void)
{
    expression();
    if(sym == eql || sym == neq || sym == lss || sym == leq || sym ==
    grt || sym == geq)
    {
        getsym();
        expression();
    }
    else
    {
        error
    }
}
```



Statement

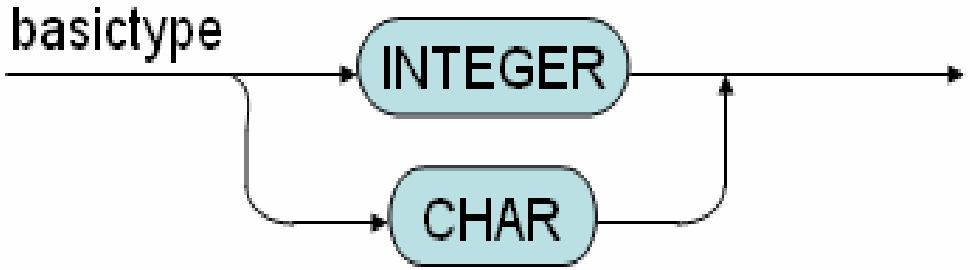

statement
assignst
variable
:=
expression
proccall
CALL
procedureidentifier
(
,
expression
)
groupst
BEGIN
;
statement
END
ifst
IF
condition
THEN
statement
whilest
WHILE
condition
DO
statement
forst
FOR
variableidentifier
:=
expression
TO
expression
DO
statement
emptyst

Compile statement function

```
void statement(void)
{
        if(accept(ident))
        {
                expect(becomes);
                expression();
                // variable :=
                                }
        else if(accept(callsym))
        {
                expect(ident);
                expect(lparen);
                expression();
                while (sym == comma)
                {
                                getsym();
                                expression();
                }
                expect(rparen);
        }
        else if(accept(beginsym))
        {
                statement();
                while(sym == semicolon)
                {
                                getsym();
                                statement();
                }
                expect(endsym);
        }
        else if(accept(ifsym))
        {
                condition();
                expect(thensym);
                statement();
                if (accept(elsesym))
                                statement();
        }
        else if(accept(whilesym))
        {
                condition();
                expect(dosym);
                statement();
        }
        else if (accept(forsym))
        {
                expect(ident);
                expect(becomes);
                expression();
                expect(tosym);
                expression();
                expect(dosym);
                statement();
        }
        else
        {
                getsym();
        }
}
```

Compile basic type function

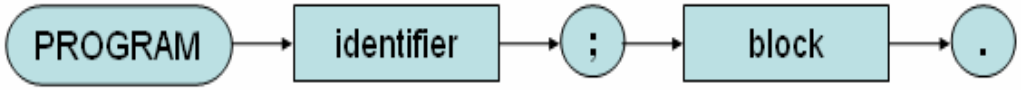
```
void basictype()
{
    if(accept(integersym)){}
    else
      expect(charsym);
}
```



Compile program function

```
void program()
{
        expect(programsym);
        expect(ident);
        expect(semicolon);
        block();
        if(sym == period)
        {
            printf("No error!"
            return;
        }
        else
        {
            error("Syntax error.");
        }
}
```

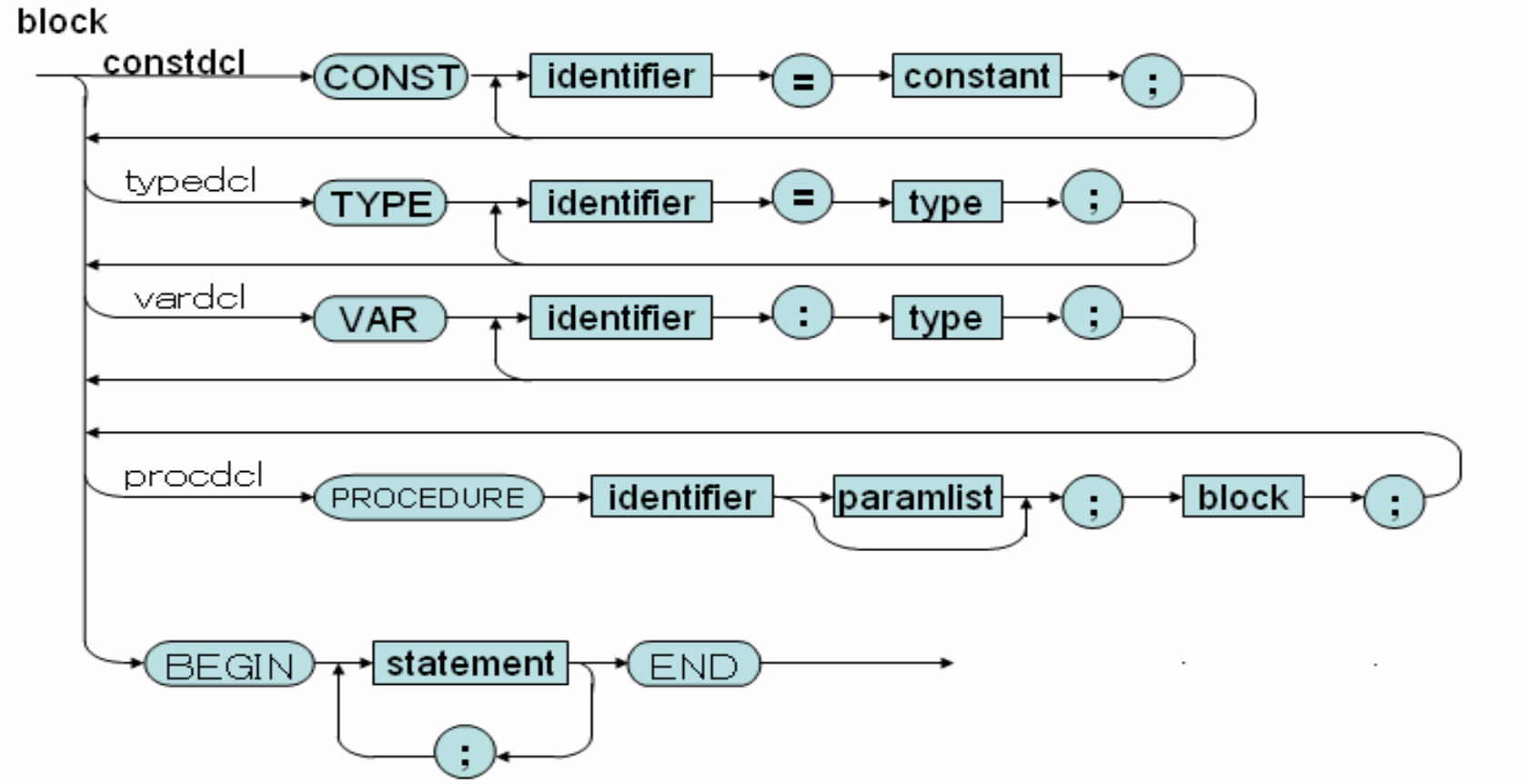
program



Compile block function

```
void block(void)
{
        if(accept(constsym)) // const
        {
            while (accept(ident))
            {
                        expect(eql);
                        constant_decl();
                        expect(semicolon);
            }
        }
        if (accept(typesym)) // type
        {
            while (accept(ident))
            {
                        expect(eql);
                        type();
                        expect(semicolon);
            }
        }
        if(accept(varsym))
            {
                while (accept(ident))
                {
                        expect(colon);
                        type();
                        expect(semicolon);
                }
            }
            while(sym == procsym)
            {
                getsym();
                expect(ident);
                if (accept(lparen))
                {
                        paramlist();
                        expect(rparen);
                }
                expect(semicolon);
                block();
                expect(semicolon);
            }
                expect(beginsym);
            statement();
            while(accept(semicolon))
                statement();
            expect(endsym);
}
```

Block


block
constdcl
CONST
identifier
=
constant
;
typedcl
TYPE
identifier
=
type
;
vardcl
VAR
identifier
:
type
;
procdcl
PROCEDURE
identifier
paramlist
;
block
;
BEGIN
statement
END
;