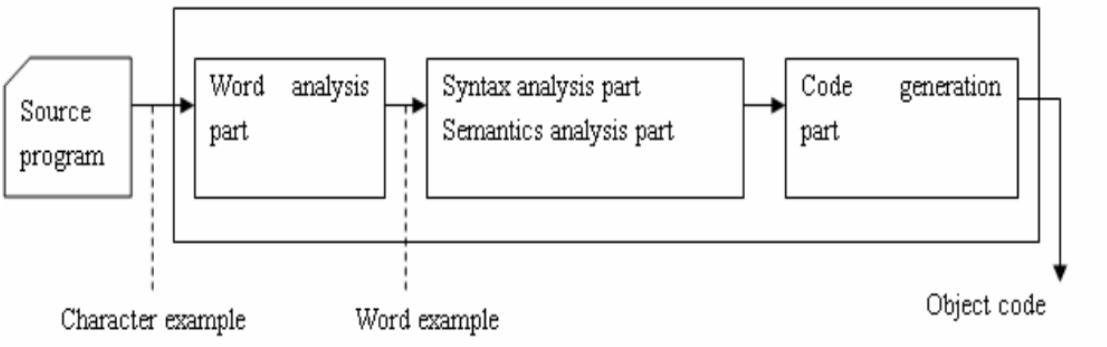
Table of Contents

1 Overview

1.1 Purpose of compiler

In simple words, A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for converting a source code is to create an executable program. Another critical goal of a compiler is to report errors in source code to developers.

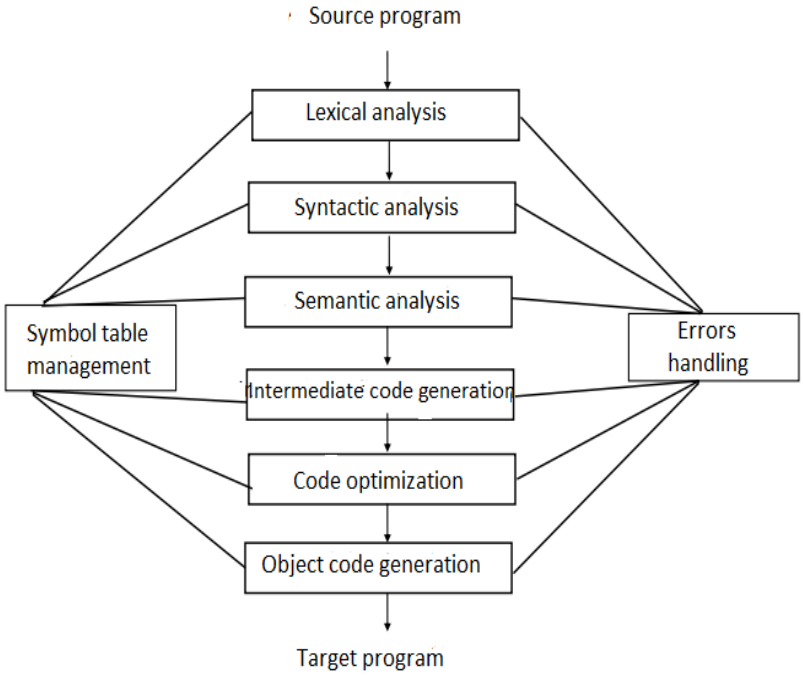
1.2 Stages in compiler



A typical compiler can be divided into 4 main parts:

- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Code generator

1.3 Main phases of compiling process



2 Lexical analysis (scanner)

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors and then breaks these syntaxes into a series of tokens. In a compiler, the program that perform lexical analysis is called the scanner.

2.1 Task of a scanner

- Skip meaningless characters: blank, tab, new line character, comment.
- Recognize illegal character and return error message.
- Recognize different types of token
    - Identifier
    - Keyword
    - Number
    - Special character
    - ….
- Pass recognized tokens to the parser to perform job of syntactic analysis.

2.2 KPL's tokens:

- Keywords: PROGRAM, CONST, TYPE, VAR, PROCEDURE, FUNCTION, BEGIN, END, ARRAY, OF, INTEGER, CHAR, CALL, IF, THEN, ELSE, WHILE, DO, FOR, TO
- Operators:
  := (assign),
  + (addition), - (subtraction), * (multiplication), / (division)
  = (comparison of equality), != (comparison of difference), > (comparison of greaterness), < (comparison of lessness), >= (comparison of greaterness or equality), <= (comparison of lessness or equality)
- Special characters: ; (semicolon), . (period), : (colon), , (comma), ((left parenthesis),) (right parenthesis), ‘ (singlequote), (. and .) to mark the index of an array, element (* and *) to mark the comment
- Others: identifier, number, illegal character

2.3 Main functions of a lexical analyzer

- void skipComment(): skip comment (string that start by (* and end by *))

```
void skipComment() {
  // TODO
  int state = 0;
  while ((currentChar != EOF) && (state < 2)) {
    switch (charCodes[currentChar]) {
    case CHAR_TIMES:
      state = 1;
      break;
    case CHAR_RPAR:
      if (state == 1) state = 2;
      else state = 0;
      break;
    default:
      state = 0;
    }
    readChar();
  }
  if (state != 2)
    error(ERR_ENDOFCOMMENT, lineNo, colNo);
}
```

- Token *readIdentKeyword(void): if the first character of string is alphabet, it can be an identifier or keyword.

```
Token* readIdentKeyword(void) {
  // TODO
  Token *token = makeToken(TK_NONE, lineNo, colNo);
  int count = 1;

  token->string[0] = (char)currentChar;
  readChar();

  while ((currentChar != EOF) &&
   ((charCodes[currentChar] == CHAR_LETTER) || (charCodes[currentChar] == CHAR_DIGIT))) {
    if (count <= MAX_IDENT_LEN) token->string[count++] = (char)currentChar;
    readChar();
  }

  if (count > MAX_IDENT_LEN) {
    error(ERR_IDENTTOOLONG, token->lineNo, token->colNo);
    return token;
  }

  token->string[count] = '\0';
  token->tokenType = checkKeyword(token->string);

  if (token->tokenType == TK_NONE)
    token->tokenType = TK_IDENT;

  return token;
}
```

3 Syntax analysis (parser)

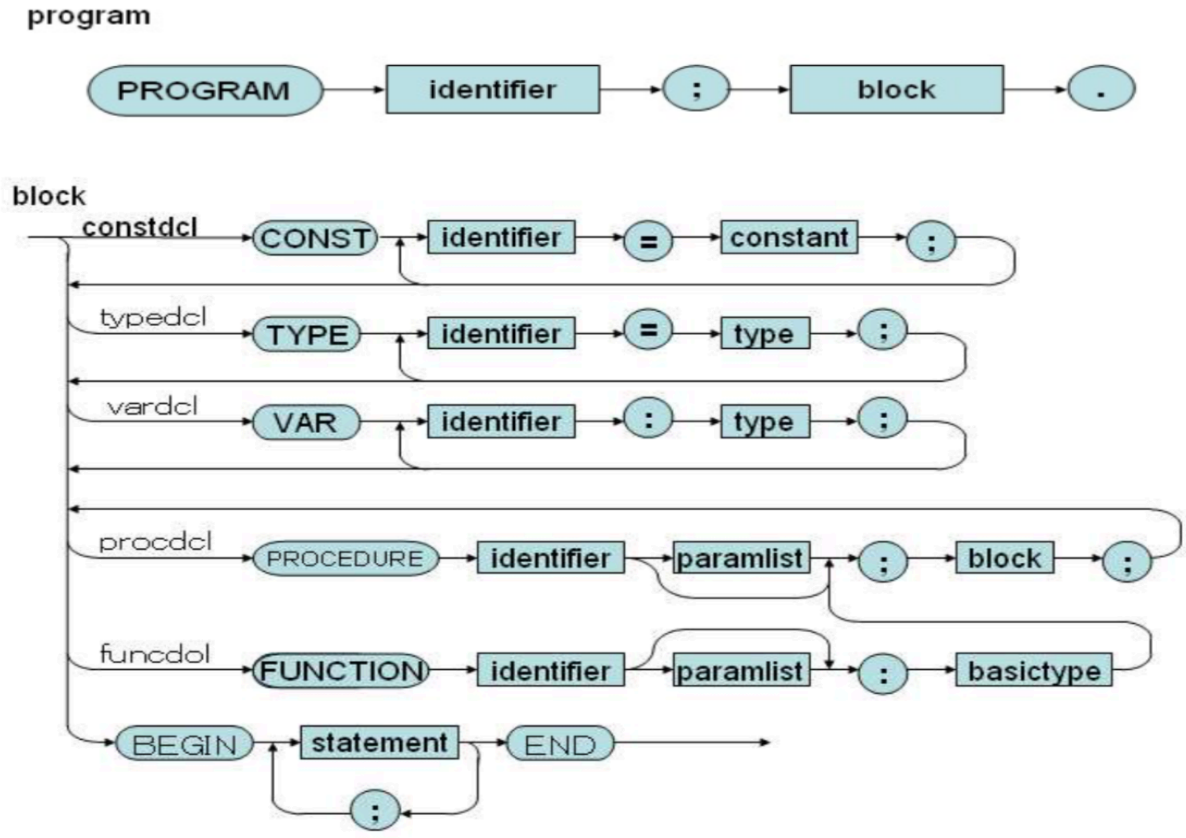
A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. It analyzes this input with the production rules and generates a parse tree as an output of this phase.

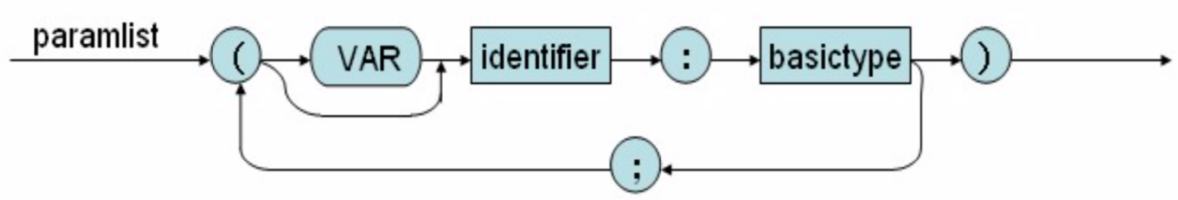
3.1 Task of syntax analysis

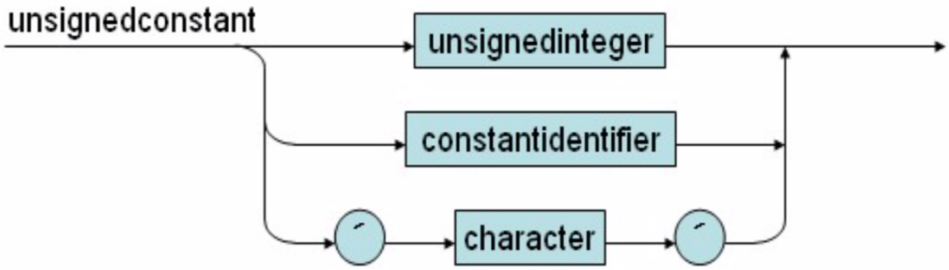
- Check the syntactic structure of a given program
- Invoke semantic analysis and code generation

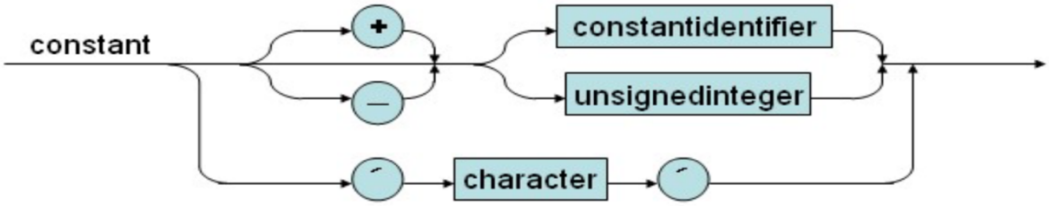
3.2 Design of syntax analysis

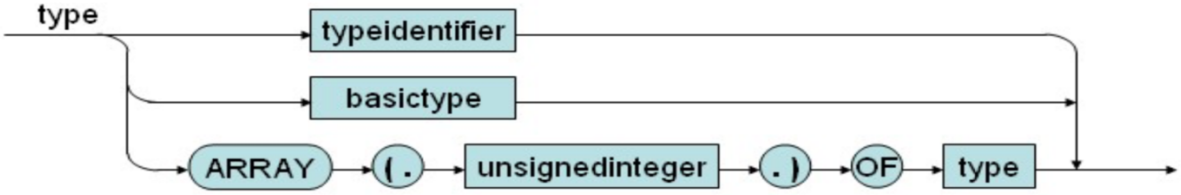
- Use top – down parsing:
  - Token LookAhead
  - Parsing terminal symbol
  - Parsing non-terminal symbol
- Use syntax diagram of KPL:



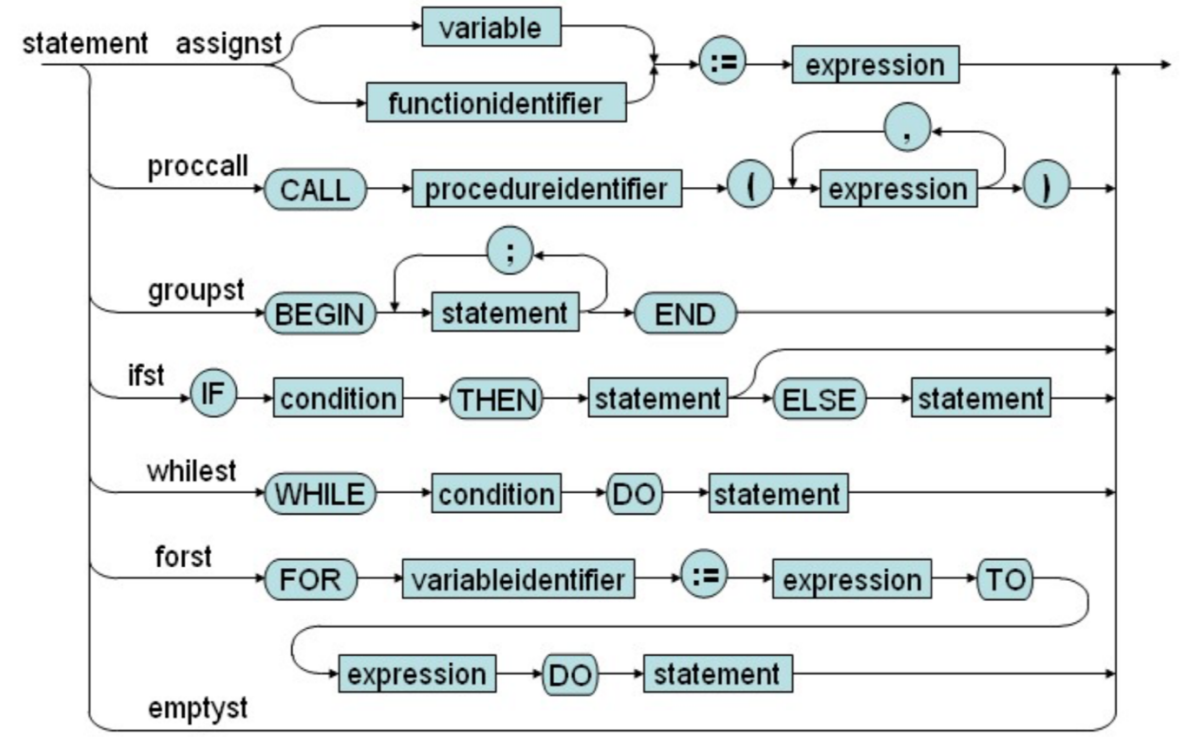

paramlist
(
VAR
identifier
:
basictype
)
;

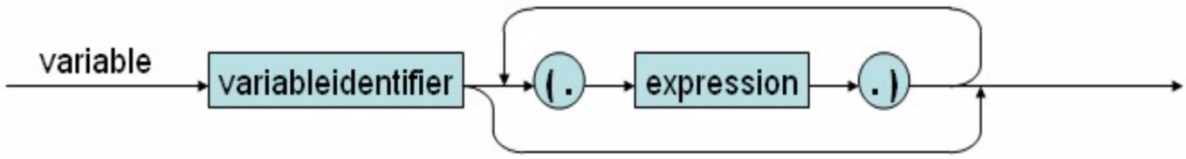

unsignedconstant
unsignedinteger
constantidentifier
'
character
'

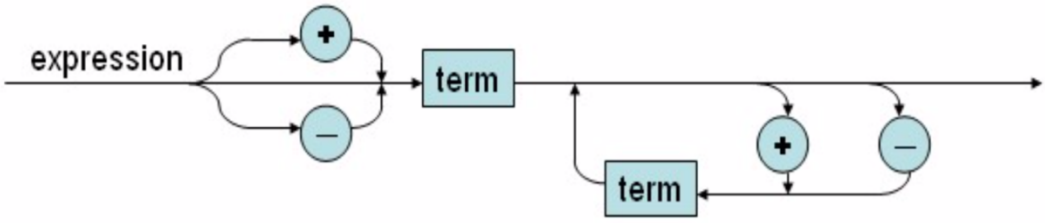

constant
+
—
constantidentifier
unsignedinteger
'
character
'

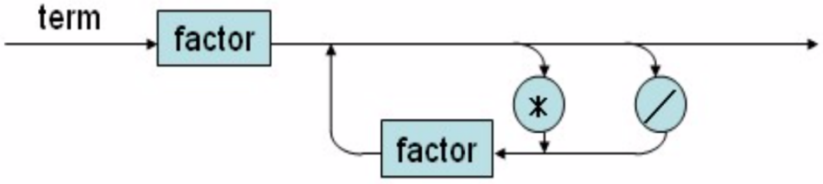

type
typeidentifier
basictype
ARRAY
(.
unsignedinteger
.)
OF
type

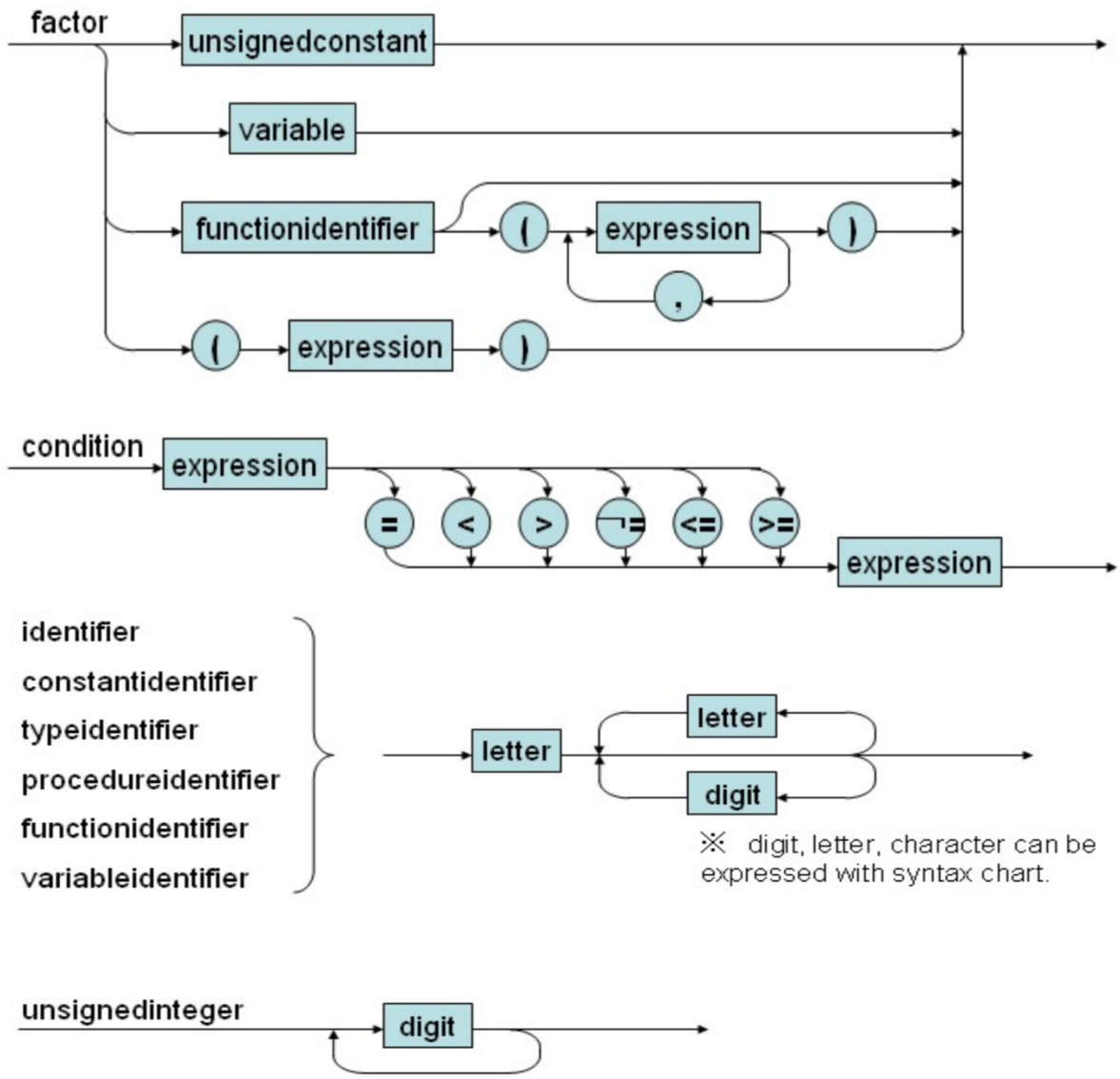

basictype
INTEGER
CHAR


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


variable
variableidentifier
(.
expression
.)


expression
+
–
term
+
–
term


term
factor
*
/
factor



3.3 Main functions of a syntax analyzer

- void compileParam (void): parsing a parameter

```
void compileParam(void) {
  // TODO
  if(lookAhead->tokenType == TK_IDENT){
    eat(TK_IDENT);
    eat(SB_COLON);
    compileBasicType();
  }else if(lookAhead->tokenType == KW_VAR){
    eat(KW_VAR);
    eat(TK_IDENT);
    eat(SB_COLON);
    compileBasicType();
  }else{
    error(ERR_INVALIDPARAM, lookAhead->lineNo, lookAhead->colNo);
  }
}
```

- void compileAssignSt (void): parsing assign statement

```
void compileAssignSt(void) {
  assert("Parsing an assign statement ....");
  // TODO
  eat(TK_IDENT);
  int countVariables = 1;
  int countExpression = 0;
  if(lookAhead->tokenType == SB_LSEL){
    compileIndexes();
  }
  while(lookAhead->tokenType == SB_COMMA){
    eat(SB_COMMA);
    eat(TK_IDENT);
    countVariables ++;
  }
  eat(SB_ASSIGN);
  compileExpression();
  countExpression++;
  while(lookAhead->tokenType == SB_COMMA){
    eat(SB_COMMA);
    compileExpression();
    countExpression++;
  }
  if(countExpression != countVariables){
    error(ERR_INVALIDSTATEMENT, lookAhead->lineNo, lookAhead->colNo);
    return;
  }
  assert("Assign statement parsed ....");
}
```

4 Semantic analysis

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment, rejecting incorrect programs or issuing warnings.

4.1 Task of semantic analysis

- Manage information about identifier (const, variable, type that user defined, sub function (function, procedure))
- Check law of semantic rule (range of identifiers, consistency of type)

4.2 Design of symbol table

- Contain information about identifier and attribute in the program:
  - Const: (identifier, type, value)
  - Type: is defined by user (identifier, real type)
  - Variable: identifier, type
  - Function: identifier, parameter, return type, local declaration

- Procedure: identifier, parameter, local declaration
- Parameter: identifier, type, value / reference

4.3 Components of symbol table

4.3.1 Definition

```
// symbol table
struct SymTab_ {
  // main program
  Object* program;

  // current scope
  Scope* currentScope;

  // Global objects such as
  // WRITEI, WRITEC, WRITELN
  // READI, READC
  ObjectNode *globalObjectList;
};
```

```
// Scope of a block
struct Scope_ {
  // List of block's objects
  ObjectNode *objList;

  // Function, procedure or program that
  //block belongs to
  Object *owner;

  // Outer scope
  struct Scope_ *outer;
};
```

- SymTab contains information about current range of variable in variable currentScope
- Each time check a procedure or a function, we need update this variable by using function void enterBlock(Scope *scope)
- In the end, we need to change current range by void exitBlock(void)
- To create a new range, we use:
  - Scope *createScope(Object *owner, Scope *outer)

4.3.2 Object

```
// Object
// classification

enum ObjectKind {
  OBJ_CONSTANT,
  OBJ_VARIABLE,
  OBJ_TYPE,
  OBJ_FUNCTION,
  OBJ_PROCEDURE,
  OBJ_PARAMETER,
  OBJ_PROGRAM
};
```

```
// Objects' attributes in symbol
// table

struct Object_ {
  char name[MAX_IDENT_LEN];
  enum ObjectKind kind;
  union {
    ConstantAttributes* constAttrs;
    VariableAttributes* varAttrs;
    TypeAttributes* typeAttrs;
    FunctionAttributes* funcAttrs;
    ProcedureAttributes* procAttrs;
    ProgramAttributes* progAttrs;
    ParameterAttributes* paramAttrs;
  };
};
```

4.3.3 Type and constant

```
// Type classification

enum TypeClass {
  TP_INT,
  TP_CHAR,
  TP_ARRAY
};

struct Type_ {
  enum TypeClass
   typeClass;

  // Use for type Array
  int arraySize;

  struct Type_
   *elementType;
};
```

```
// Constant

struct ConstantValue_ {
  enum TypeClass type;
  union {
    int intValue;
    char charValue;
  };
};
```

4.4 Semantic rule

4.4.1 Check identifier

- void checkFreshIdent(char* name): check this identifier existing in current

4.4.2 Check declaration of identifier

- Object *checkDeclaredIdent(char *name);
- Object *checkDeclaredConstant(char *name);
- Object *checkDeclaredType(char *name);
- Object *checkDeclaredVariable(char *name);
- Object *checkDeclaredType(char *name);
- Object *checkDeclaredFunction(char *name);
- Object *checkDeclaredProcedure(char *name);
- Object *checkDeclaredLValueIdent (char *name);

4.4.3 Check consistency between defined identifiers and using identifier:

- void checkIntType(Type *type): check integer type
- void checkCharType(Type *type): check char type
- void checkArrayType(Type *type): check array type
- void checkBasicType(Type *type): check basic type
- void checkTypeEquality(Type *type1, Type *type2): compare two type

5 References:

- Lecture slide: Compiler construction 2019 – 2020 – Dr. Nguyen Thi Thu Huong