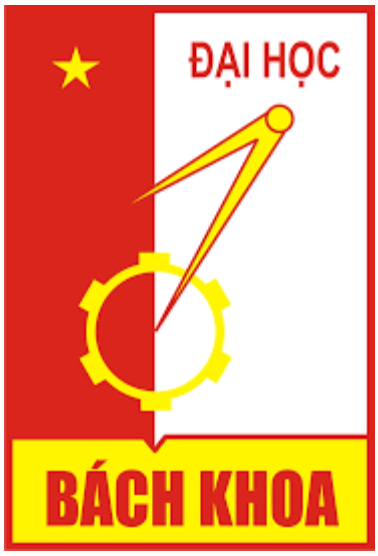
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

----------o0o----------



COMPILER CONSTRUCTION REPORT

Student: Nguyen Binh Long – 20176807
Instructor: Dr. Nguyen Thi Thu Huong

Hanoi, January 2020

TABLE OF CONTENTS

CHAPTER 1: AN OVERVIEW OF COMPILER

1.1. TASK OF A COMPILER

A compiler is a computer program (or set of programs) that transforms source code written in a programing language (the source language) into a machine language (the target language). The most common reason for converting a source code into object code is to create an executable program. Another goal of a compiler is to report errors in source code to programmer.

(see more at: http://en.wikipedia.org/wiki/Compiler)

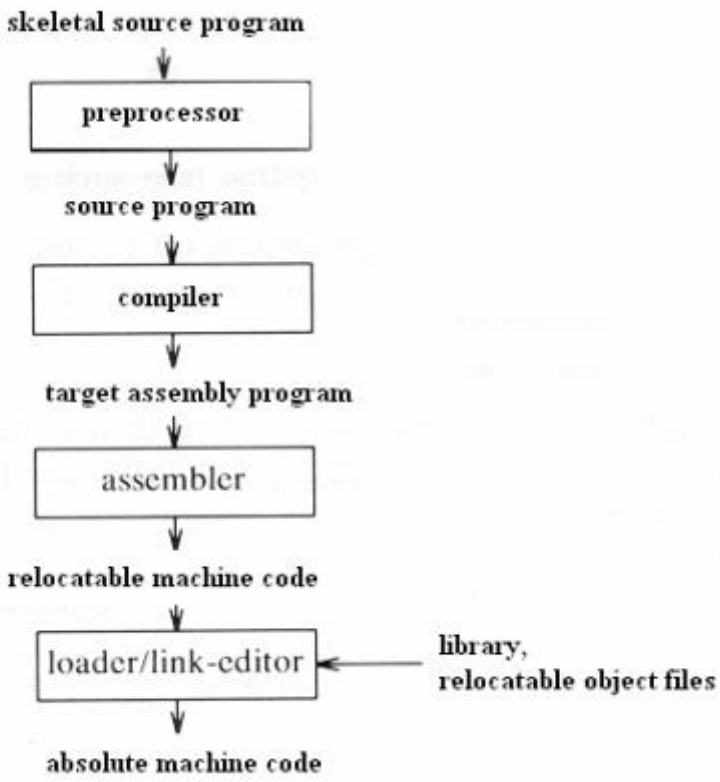


Figure 1: Compiler in a language processing system

1.2. COMPONENTS OF A COMPILER

A typical compiler can be divided into 4 main parts:

- Lexical analyzer: Scan characters in source code and combine them into group of significant words called token
- Syntax analyzer: Check syntactic structure of a given program
- Semantic analyzer: Check the source program to find semantic errors and collect information to build code generation stage.

- Code generator: Generate destination code that includes machine code and assembly

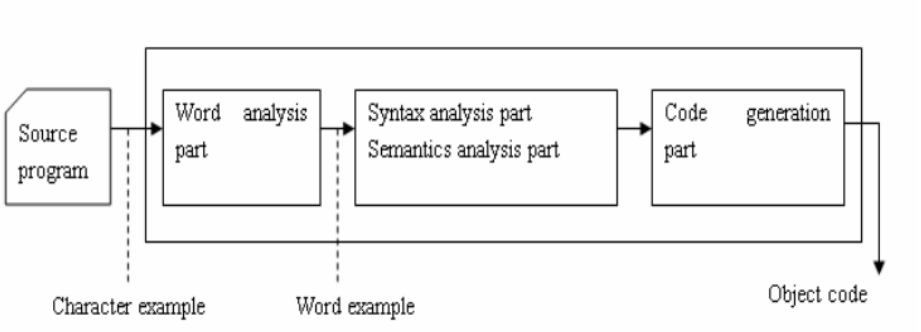


Figure 2: Components of a compiler

1.3. MAIN PHASES OF COMPILING PROCESS

To describe more precisely, a compiler is divided into several interrelated processes, in each process, source program is translated from a specific form to another form of representation. An example of typical decomposition is illustrated in the following figure:

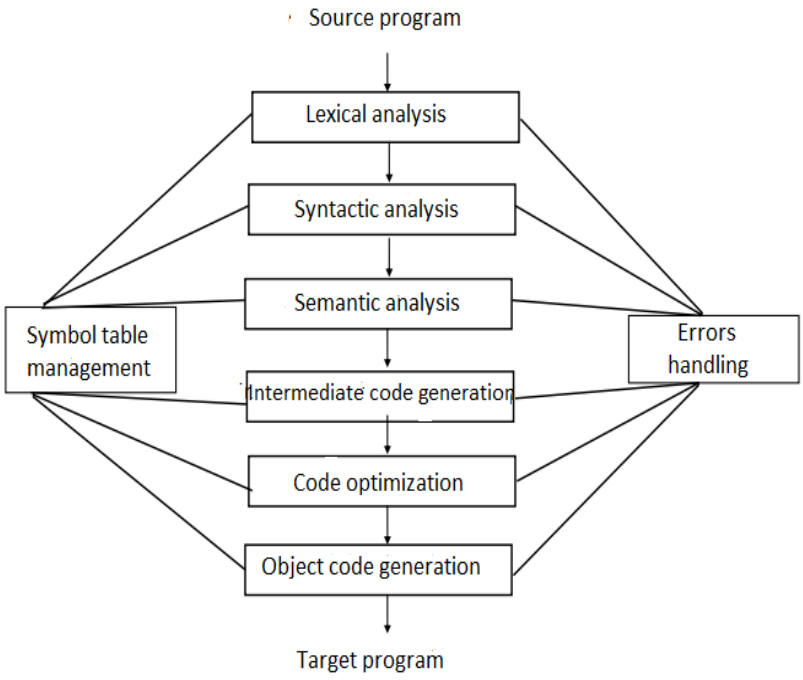


Figure 3: Main phases in the compiling process

1.3.1. Lexical analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens, i.e. meaningful character strings. A program or function that performs lexical analysis is called a lexical analyzer, lexer, tokenizer, or scanner.

1.3.2. Syntactic analysis

Syntactic analysis (also called parsing) is the process of analysing a string of tokens, conforming to the rules of a formal grammar or not. The program that performs parsing is called the syntactic analyser or simply parser.

1.3.3. Semantic analysis

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 type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings.

1.3.4. Intermediate code generation

After the phase of semantic analysis, some compiler will generate an intermediate representation of source program, known as intermediate code which has 2 important properties: easy to generate, and easy to translate into object code. Moreover, intermediate code is machine-independent.

1.3.5. Code optimization

In this phase, code optimizator will try to optimize the intermediate code into equivalent one with faster execution.

1.3.6. Object code generation

This is the final phase of compiler. Input of a object code generator is the intermediate code and output is the target program.

1.4. SUMMARY

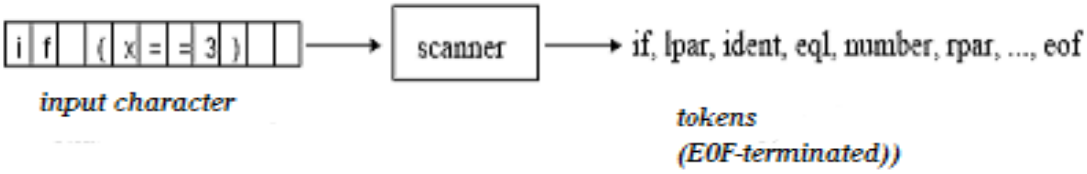
In order for a computer to understand and execute a program written in a high-level programming language, we need a compiler to translate source program into target program in object codes. This chapter has presented an overview of a compiler, including lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, code optimization and object code generation. In general, the output of the preceding phases are input of the next phases in the compiler.

CHAPTER 2: DESIGN A LEXICAL ANALYZER FOR KPL

2.1. TASK OF A LEXICAL ANALYSER (SCANNER)

- Neglect meaningless character: space, tabulalor, EOF, CR, LF, comments.
- Detect invalid symbols: @, ! (stand-alone), etc
- Detect and produce tokens: identifiers, keywords, numbers, literals, special characters, etc.

For example:



2.2. TOKENS IN KPL

- Identifier: variable, constant, type, function, procedure:
  - Start with letter or underscore: a-z, A-Z, '_'
  - Others are letter, underscore or numbers
- Keywords: PROGRAM, CONST, TYPE, VAR, PROCEDURE, FUNCTION, BEGIN, END, ARRAY, OF, INTEGER, CHAR, CALL, IF, 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 specify indexes in arrays, (*, *) to indicate comments
- Others: integer number, string literals…

2.3. DATA STRUCTURES IN SCANNER FOR KPL

typedef enum {

CHAR_SPACE, CHAR_LETTER, CHAR_DIGIT, CHAR_PLUS, CHAR_MINUS, CHAR_TIMES, CHAR_SLASH, CHAR_LT, CHAR_GT, CHAR_EXCLAIMATION, CHAR_EQ, CHAR_COMMA, CHAR_PERIOD, CHAR_COLON, CHAR_SEMICOLON,

CHAR_SINGLEQUOTE, CHAR_LPAR, CHAR_RPAR,
CHAR_UNKNOWN
} **CharCode;**

⇨ to store valid characters in KPL : space, letters, numbers, +, -, *, /, <, >, ! , … others are invalid characters (CHAR_UNKNOWN)

typedef enum {
TK_NONE, TK_IDENT, TK_NUMBER, TK_CHAR, TK_EOF,
KW_PROGRAM, KW_CONST, KW_TYPE, KW_VAR,
KW_INTEGER, KW_CHAR, KW_ARRAY, KW_OF,
KW_FUNCTION, KW_PROCEDURE,
KW_BEGIN, KW_END, KW_CALL,
KW_IF, KW_THEN, KW_ELSE,
KW_WHILE, KW_DO, KW_FOR, KW_TO,
SB_SEMICOLON, SB_COLON, SB_PERIOD, SB_COMMA,
SB_ASSIGN, SB_EQ, SB_NEQ, SB_LT, SB_LE, SB_GT, SB_GE,
SB_PLUS, SB_MINUS, SB_TIMES, SB_SLASH,
SB_LPAR, SB_RPAR, SB_LSEL, SB_RSEL
} **TokenType;**

⇨ to store token types in KPL

typedef struct {
char string[MAX_IDENT_LEN + 1];
int lineNo, colNo; // line and column of tokens
TokenType tokenType;
int value;
} Token;

⇨ to store information about each tokens:
- string : content of token
- lineNo , colNo : position of token,
- tokenType : type of token
- value : value of token if a number.

2.4. FUNCTIONS IN SCANNER FOR KPL

2.4.1. Details about functions

void *skipBlank*() : skip spaces.
void *skipComment*() : skip comments.
Token* *readIdentKeyword*() : read identifiers/keywords, return a pointer of Token type.
Token* *readNumber*() : read a integer number, return a pointer of Token type.

Token* *readConstChar*() : read a constant character, return a pointer of Token type.

TokenType *checkKeyword*(char *string) : check if the string is a keyword, return TOKEN_NONE if keyword.

Token* *makeToken*(TokenType tokenType, int lineNo, int colNo) : create a pointer to a token with predefined type and position.

Token* *getToken*() : read and return a token (can be invalid token: TOKEN_NONE).

Token* *getValidToken*() : read and return a valid token.

2.4.2. Details about execution of a scanner.

Scanner is a finite automation. During the reading of input stream, getToken() function is called recursively to determine the next token.

Details about implementation of functions, refer to file "scanner.c".

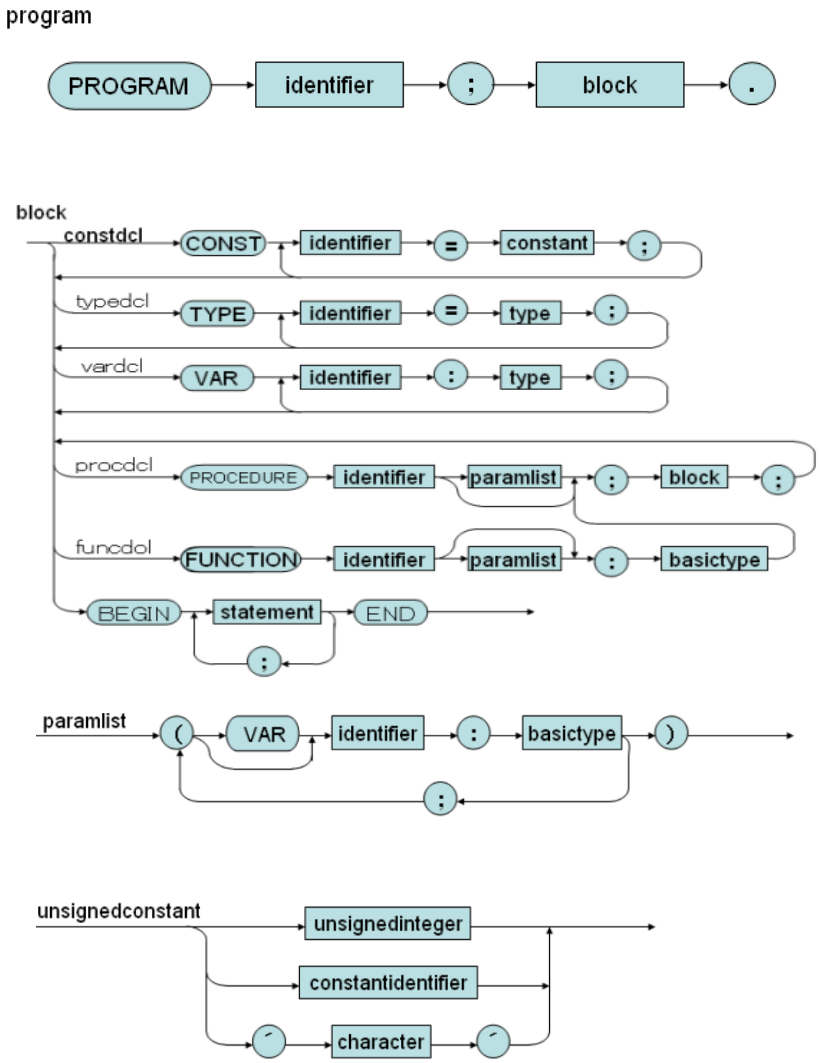
CHAPTER 3: DESIGN A SYNTACTIC ANALYSER FOR KPL

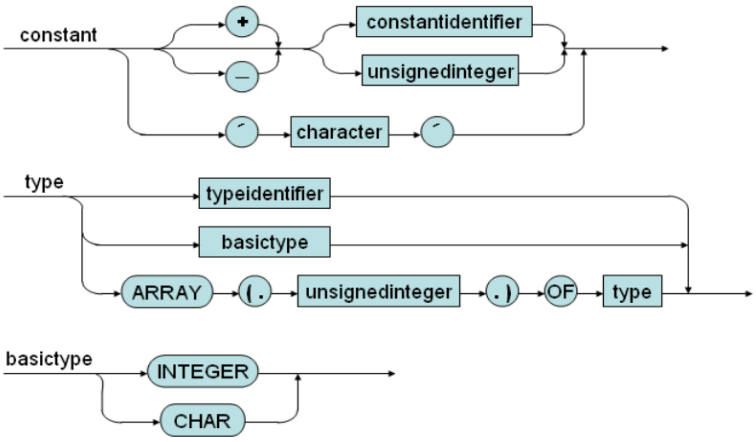
3.1. TASK OF A SYNTATIC ANALYSER (PARSER)

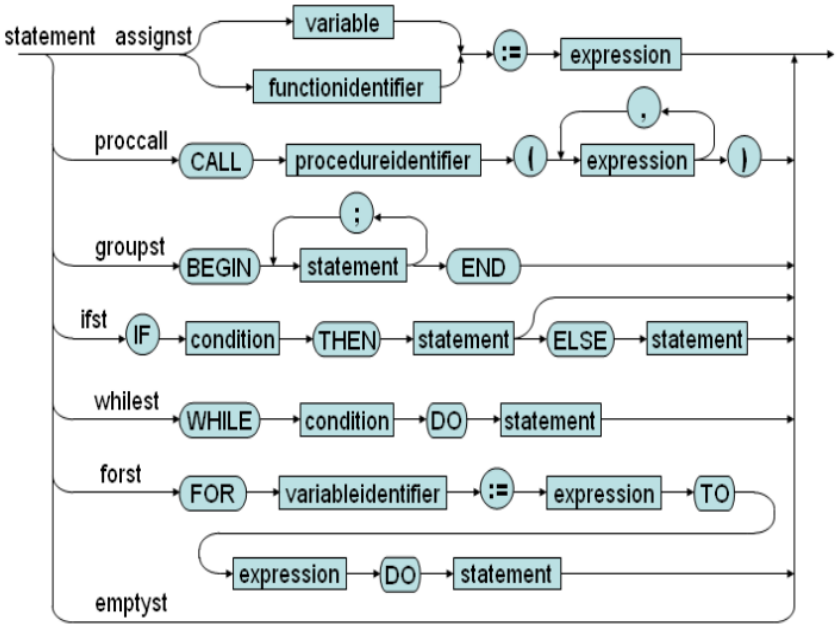
- Check the syntax of the program for errors.
- Produce parse tree for semantic analyser otherwise.

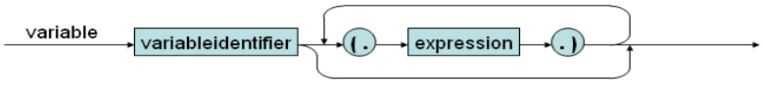
3.2. SYNTAX DIAGRAM AND BNF GRAMMAR

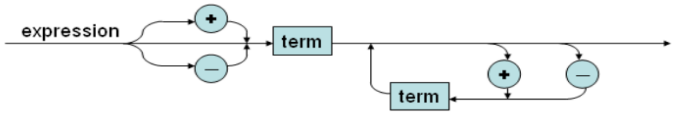
3.2.1. Syntax diagram.

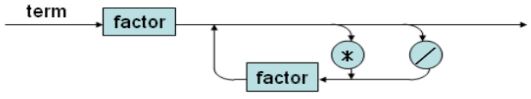


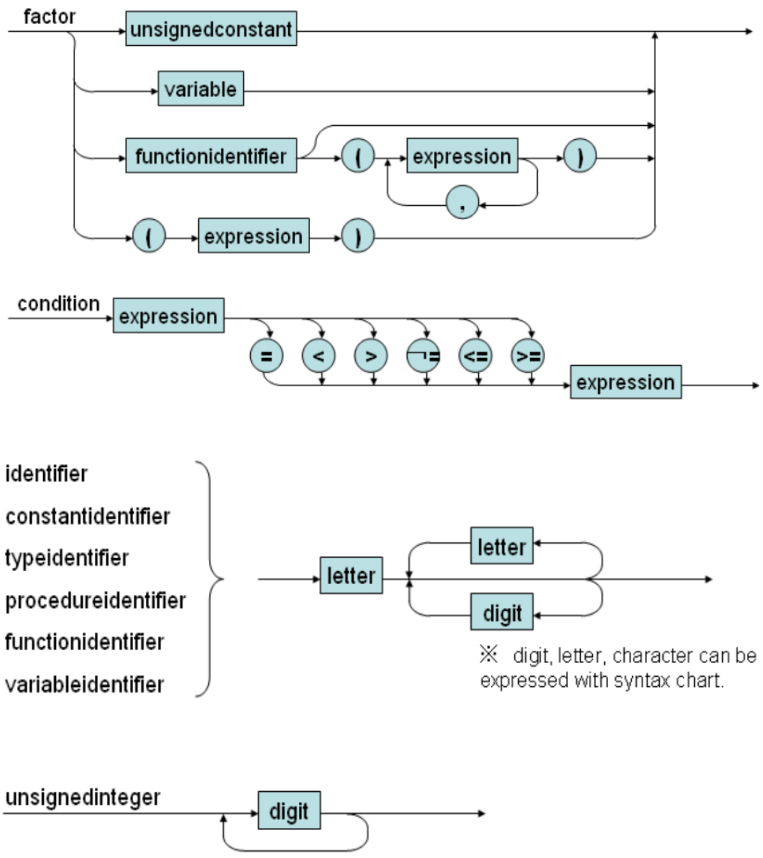

constant
+
-
constantidentifier
unsignedinteger
'
character
'
type
typeidentifier
basictype
ARRAY
(.
unsignedinteger
.)
OF
type
basictype
INTEGER
CHAR


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



3.2.2. BNF grammar.

Backus–Naur form or Backus normal form (BNF) is a notation technique for context-free grammars, often used to describe the syntax of languages.

Below are some of the BNF grammar in KPL:

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

3.3. RECURSIVE DESCENT PARSING

- Properties:
  - LL(k) is the language that lools ahead k character to produce a valid production.
  - Used to parse LL(1) language.
  - Can be extended for LL(k), but very complex.
  - Used for other grammar can lead to infinite iteration.
- Recursive descent parsing:
  - A top-down parsing method.
  - The term descent refers to the direction in which the parse tree is traversed.
  - 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.

3.4. DATA STRUCTURES IN PARSER FOR KPL

- Use data structure in scanner.

3.5. PARSING TERMINAL AND NON-TERMINAL SYMBOLS

3.5.1 Terminal Symbols

void *eat*(TokenType tokenType);

- Function will compare the passed tokenType to token type read in scanner (currentToken).
- If equals, print out the token. Otherwise, report error: “missing token” at that position.

3.5.2 Non-terminal Symbols

void *compileProgram*(): parse main program.
void *compileBlock*(void): parse constant declarations then call *compileBlock2*.
void *compileBlock2*(void): parse type declarations then call *compileBlock3*.
void *compileBlock3*(void): parse variable declarations then call *compileBlock4*.
void *compileBlock4*(void):parse subroutines declarations then call *compileBlock5*.
void *compileBlock5*(void): parse statements in main function.
void *compileConstDecls*(void): parse constant declarations.
void *compileConstDecl*(void): parse a single constant declaration.
void *compileTypeDecls*(void): parse type declarations.
void *compileTypeDecl*(void): parse a single type declaration.
void *compileVarDecls*(void): parse variable declarations.
void *compileVarDecl*(void): parse a variable declaration.
void *compileSubDecls*(void): parse subroutines declarations.
void *compileFuncDecl*(void): parse function declarations.

void *compileProcDecl*(void): parse procedures declarations.
void *compileUnsignedConstant*(void): parse unsigned constants.
void *compileConstant*(void): parse signed constants.
void *compileType*(void): parse a type.
void *compileBasicType*(void): parse a basic type.
void *compileParams*(void): parse list of parameters.
void *compileParam*(void): parse a single parameter.
void *compileStatements*(void): parse all statements.
void *compileStatement*(void): parse a single statement.
void *compileAssignSt*(void): parse an assignment statement.
void *compileCallSt*(void): parse a call statement.
void *compileIfSt*(void): parse an IF statement.
void *compileElseSt*(void): parse an ELSE statement.
void *compileWhileSt*(void): parse a WHILE statement.
void *compileForSt*(void): parse a FOR statement.
void *compileArguments*(void): parse list of arguments passed to a function or procedure.
void *compileArguments2*(void): parse list of arguments passed to a function or procedure.
void *compileCondition*(void): parse conditional expression.
void *compileExpression*(void): parse (+,-) of an expression then call *compileExpression2*
void *compileExpression2*(void): parse (+, -) operators between terms then call *compileExpression3*
void *compileExpression3*(void): recursive of (+, -) operators between terms.
void *compileTerm*(void): compile a term, which can be composed of (*, /) of compileFactor
void *compileTerm2*(void) recursive procedure of (*, /) between factors.
void *compileFactor*(void): a factor can be a number, character, identifier .
void *compileIndexes*(void): parse indexes of an array.

Details about implementation of functions, refer to "parser.c"

CHAPTER 4: DESIGN A SEMANTIC ANALYZER FOR KPL

4.1. TASK OF A SEMANTIC ANALYZER

- Produce symbol table to manage attributes of identifier (const, variable, type, function, procedure)
- Check semantic rules: Scope checking (range of identifiers) and Type checking (consistency of types)

4.2. DESIGN A SYMBOL TABLE

4.2.1. Symbol table

We need a symbol table to store information needed about every identifiers in the program. Each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and its location.

4.2.2. Design symbol table

Contain information about identifier and attribute in the program:

- Constance: (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

4.2.3. Data structure in symbol table

```
struct SymTab_ {
  Object* program;
  Scope* currentScope;
  ObjectNode *globalObjectList;
};
```

The data structure to represent symbol table itself, including:

- Program: the program object.
- currentScope: current scope of symbol table.
- globalObjectList: store global objects such as functions: CALLI, WRITEI, etc.

```
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;
  };
};
```

⇨ To store information about each object in program, such as main program itself, a procedure or function, a variable, a constant, etc.

```
struct Scope_ {
  ObjectNode *objList;
  Object *owner;
  struct Scope_ *outer;
};
```

⇨ To store information about a scope, including:

- objList: objects in the scope.
- owner: function or procedure has that scope.
- outer: the outer scope of it.

4.2.4. Functions in symbol table.

Object* *createProgramObject*(char *programName): create a program object.
Object* *createConstantObject*(char *name): create a constant object.
Object* *createTypeObject*(char *name): create a type object.
Object* *createVariableObject*(char *name): create a variable object.
Object* *createFunctionObject*(char *name): create a function object.
Object* *createProcedureObject*(char *name): create a procedure object.
Object* *createParameterObject*(char *name, enum ParamKind kind, Object* owner): create a parameter object.

Type* *makeIntType*(void): create an integer type.
Type* *makeCharType*(void): create a character type.
Type* *makeArrayType*(int arraySize, Type* elementType): create an array type.
Type* *duplicateType*(Type* type): copy type.
int *compareType*(Type* type1, Type* type2): compare type

ConstantValue* *makeIntConstant*(int i): create an integer constant.
ConstantValue* *makeCharConstant*(char ch): create a character constant.
ConstantValue* *duplicateConstantValue*(ConstantValue* v): copy a constant.

Scope* *createScope*(Object* owner, Scope* outer): create a scope.
Object* *findObject*(ObjectNode *objList, char *name): find object with specific name in an object list.

4.3. CHECK SEMANTIC RULES

4.3.1. Checking fresh identifier

void *checkFreshIdent*(char *name): to determine if an identifier is declared or not.

4.3.2. Checking declared identifier

Object* *checkDeclaredIdent*(char *name): check declared identifiers: (identifier is already declared or not. If declared return identifier object, else return null.)
Object* *checkDeclaredConstant*(char *name): check declared constants.
Object* *checkDeclaredType*(char *name): check declared identifiers.
Object* *checkDeclaredVariable*(char *name): check declared variables.
Object* *checkDeclaredFunction*(char *name): check declared functions.
Object* *checkDeclaredProcedure*(char *name): check declared procedure.
Object* *checkDeclaredLValueIdent*(char *name): check declared LValue.

4.3.3. Checking the consistency between declaration and usage of identifiers.

void *checkIntType*(Type* type): check if type is integer
void *checkCharType*(Type* type): check if type is character
void *checkArrayType*(Type* type): check if type is array type.
void *checkBasicType*(Type* type): check if type is basic type.
void *checkTypeEquality*(Type* type1, Type* type2): check for equality of types, if not, report an error message.

REFERENCES

1) Dr. Nguyen Thi Thu Huong, *Compiler Contruction Lecture Slides*, 2019-2020.
2) Alfred V. Aho, Monica S.Lam, Ravi Sethi, Jeffrey D.U, *Compiler : Principles, Techniques and Tools*, 2nd edition – 2007
3) https://www.wikipedia.org/