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

========

COMPILER CONSTRUCTION REPORT

Under the supervision of
Dr. NGUYỄN THỊ THU HƯƠNG

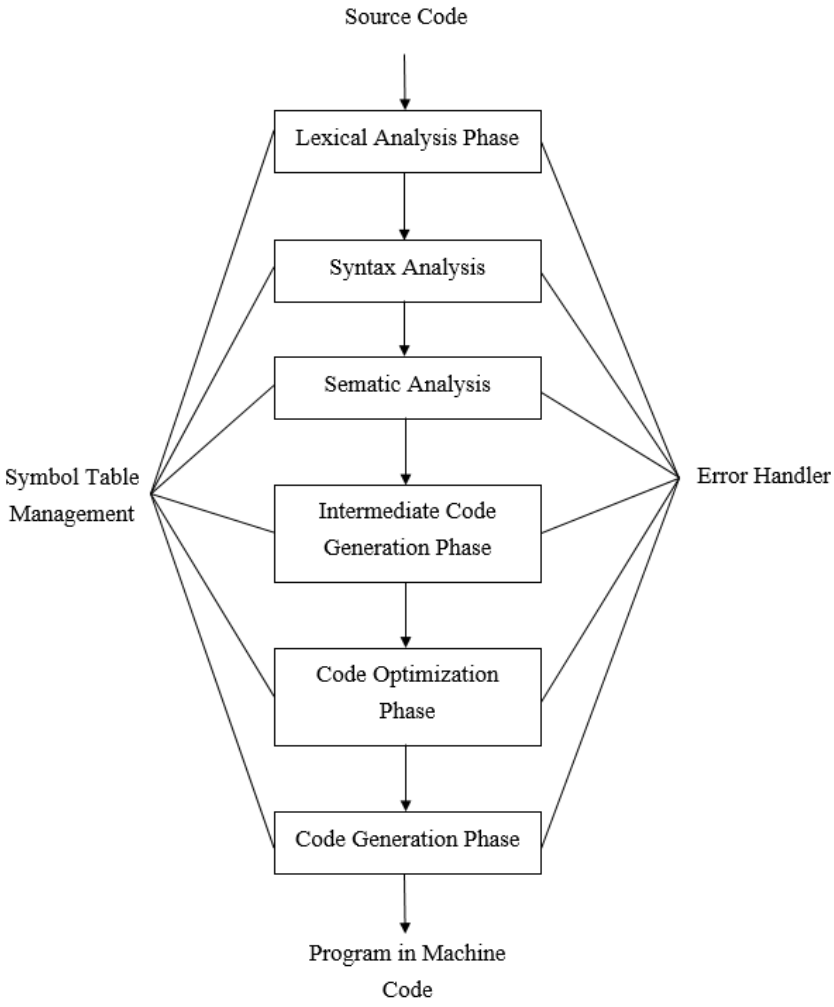
Student name: TRẦN THỊ HẰNG
Student ID: 20176748

Hanoi, January 2021

Table of contents

1. Overview

1.1 Main component of a compiler



A typical compiler can be divided into 4 main parts:

- Lexical analyzer: Breakdown input source code into a sequence of tokens of lexemes for consummation later.
- Syntax analyzer: Checks syntactic structure of a given program.
- Semantic analyzer: Checks source program to find semantic errors and collect information to build code generation stage.
- Code generator: Generates destination code that includes machine and assembly.

2. Lexical analysis design

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace, comments or redundant characters in the source code.

If the lexical analyzer finds an invalid token, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams

from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

In this part, I have to complete some important functions in file ***scanner.c***

- void skipBlank();
- void skipComment();
- Token* readIdentKeyword(void);
- Token* readNumber(void);
- Token* readConstChar(void); •Token* getToken(void);

2.1 Scanner tasks

- Skip meaningless characters: blank, tab, new fine 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 syntax analysis.

2.2 Tokens of KPL

Identifiers

Variable names, constant names, type names, function names, procedure names starting with letter or underscore followed by letters, underscores 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, = equality comparison, ! difference comparison, > greater comparison, >= greater or equal comparison, < less comparison, <= less or equal comparison

Special characters

; semicolon, . period, : colon, , comma, (left parenthesis, (right parenthesis, ' single quote, (. and .) to mark the index of an array element, (* and *) to mark the comment.

Others

String literals, integer number, ...

2.3 Data structure of KPL scanner

2.3.1 CharCode: Valid character type in KPL

We store list of charcodes in file ***charcode.c***, we define ***charCodes*** array that associates every ASCII character with an unique predefined CharCode to save memory space.

getc() function may return EOF (or -1) which is not an ASCII character.

2.3.2 TokenType: valid tokens in KPL

TokenType can be Token, Keyword or Symbol

2.3.3 Token structure:

char string[MAX_IDENT_LEN + 1] - content of token

int lineNo, colNo – start line and column number (position) of token

TokenType tokenType – token type (can be Token, Keyword or Symbol)

int value – value of token if it is a number

2.4 Main Functions in lexical analysis

- void skipBlank():
  Skip characters: blank character, tab, new line.
- void skipComment():
  Skip comment part (string which starts by (* and ends by *))
- Token* *readIdentKeyword*() : read identifiers or 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.

3. Syntax Analysis design

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

3.1 Parser tasks

- Parse the syntactic structure of a given program to check its structure.
- Produce parse tree (if any) for semantic analysis and trigger semantic analyzer or error message otherwise.
- If a parse tree is built succesfully, the program is grammatically correct

3.2 Recursive descent parsing

- LL(k) is the language that needs looking ahead k character to produce a valid production.
- It is used to parse LL(1) language and can be extended for parsing LL(k) grammars, but algorithms are complicated. Parsing non LL(k) grammars can cause infinite loops
- Using top - down parsing method: construct the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity.
- 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
- Consider a special type of recursive-descent parsing called predictive parsing does not require any back-tracking, use a lookahead symbol to decide which production.

This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking.

We use:

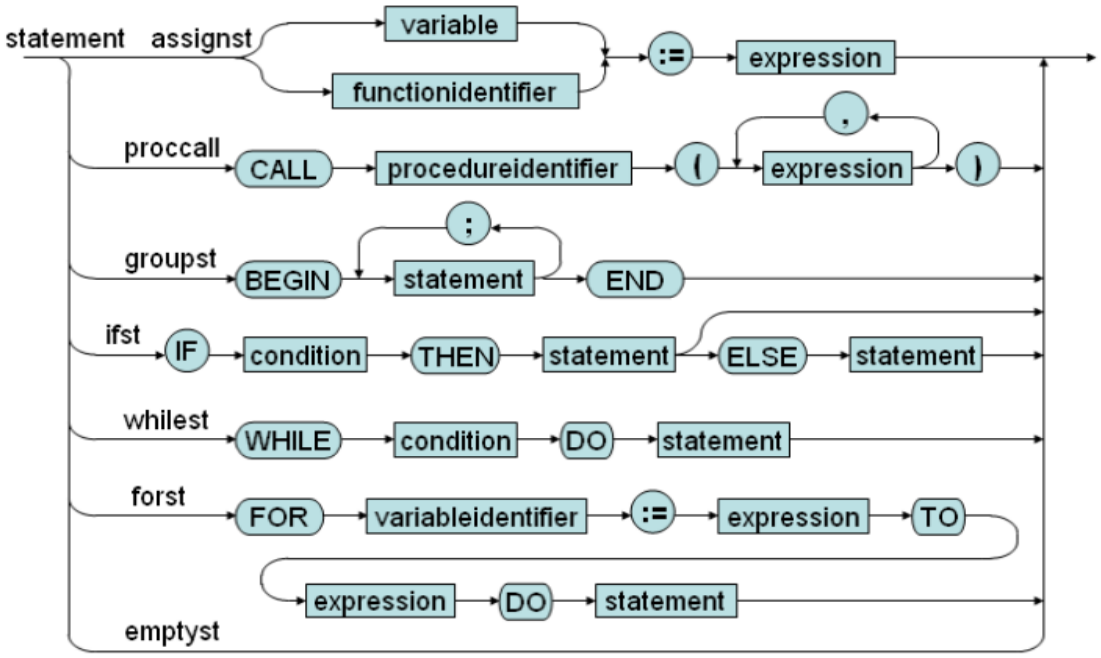
- Token lookAhead
- Parsing terminal symbol
- Parsing non-terminal symbol

3.3 Syntax diagram and BNF grammar

Backus normal form (BNF) is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols.

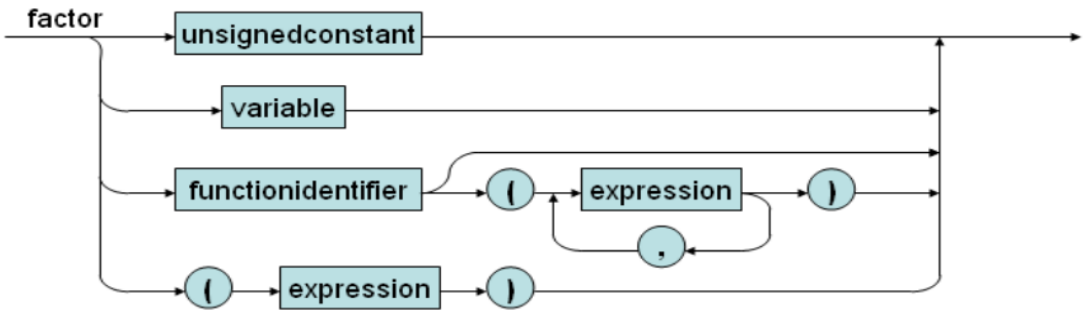
Example of set of production with Statement which is satisfied LL(1):

```
Statement ::= AssignSt
Statement ::= CallSt
Statement ::= GroupSt
Statement ::= IfSt
Statement ::= WhileSt
Statement ::= ForSt
Statement ::= ε
```



Example of set of production with Factor which is not satisfied LL(1) but LL(2):

```
Factor ::= TK_NUMBER
Factor ::= TK_CHAR
Factor ::= TK_IDENT
Factor ::= TK_IDENT Arguments
Factor ::= TK_IDENT Indexes
Factor ::= SB_LPAR Expression SB_RPAR
```



When Factor production begins with TK_IDENT, we have to look up to the next token and determine which production will be used.

If a production product **ε**, we have to calculate its **FOLLOW** set. Without FOLLOW set, a correct Program can be considered as wrong when it does not contains any block or implement; a correct statement also can be considered as INVALID_STATEMENT

3.4 Parsing terminal symbols

We use the ***eat*** function: void eat (TokenType tokenType)

We will have the lookahead compared to see if the token is terminal or not before jumping into the eat function.

3.5 Parsing non-terminal symbols

- void compileProgram( ): main program to check
- void compileBasicType( ): parsing basic type integer or char only
- void compileBlock( ), 2, 3, 4, 5: parsing block of program
- void compileConstDecls( ): parsing declaration of consts
- void compileConstDecls( ): parsing declaration of a const
- void compileTypeDecls( ): parsing declaration of types
- void compileTypeDec1( ): parsing declaration of a type
- void compileVarDecls( ): parsing declaration of variables
- void compileVarDecl( ): parsing declaration of a variable
- void compileSubDecls( ): parsing declaration of sub programs
- void compileFuncDec1( ): parsing declaration of function
- void compileProcDec1( ): parsing declaration of procedure
- void compileUnsignedConstant( ): parsing unsigned constant
- void compileConstant( ): parsing constant
- void compileType( ): parsing type
- void compileParams ( ): parsing parameters
- void compileParam ( ): parsing a parameter
- void compileStatements ( ): parsing statements
- void compileStatement ( ): parsing a statement
- void compileAssignSt ( ): parsing assign statement
- void compileCallSt ( ): parsing call statement
- void compilelfSt ( ): parsing if statement
- void cornpileElseSt ( ): parsing else statement
- void compileWhileSt ( ): parsing while statement
- void compileForSt ( ): parsing for statement
- void compileArgurnent ( ): parsing argument
- void compileArguments ( ): parsing arguments
- void compileCondition ( ): parsing condition
- void compileExpression ( ): parsing operator (+, -) and call function compileExpression2
- void compileExpression2 ( ): parsing operator (+, -) and call function compileExpression3
- void compileExpression3 ( ): parsing operator (a, -) and call function compileTerm
- void compileTerm( ): parsing term
- void compileFactor( ): Parsing a name, ident, character
- void compilelndexes(): parsing index of array

4. Semantic Analysis Design

Semantic Analysis is the third phase of Compiler. It makes sure declarations and statements of program are semantically correct. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.

Important part: Type checking and Scope management.

Semantic Analyzer uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Semantic Errors: Errors recognized by semantic analyzer are as follows:

- Type mismatch
- Undeclared variables
- Reserved identifier misuse

4.1 Design of 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.

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

4.2 Components of Symbol table

4.2.1 Definition

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

struct Scope_ {
        ObjectNode *objList;
        Object *owner;
        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( )
- To create a new range, we use:

Scope *createScope(Object *owner, Scope *outer)

4.2.2 Object

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

- Object *createProgramObject(char *programName): create new object is new program
- Object *createConstantObject(char *name): create object is constant
- Object *createTypeObject(char *name): create object is type
- Object *createVariableObject(char *name): create object is variable
- Object *createFunctionObject(char *name): create object is function
- Object *createProcedureObject(char *name): create object is procedure
- Object *createParameterObject(char *name, enum ParamKind, Object *owner): create object is parameter
- Object *findObject(ObjectNode *objList, char 'name): find an object
- void declareObject(Object *obj): declare an object
- void freeObject(Object *obj): release an object
- void freeObjectList(ObjectNode *objList): release list of object
- void freeReferenceList(ObjectNode *objList): release list of object reference

- void addObject(ObjectNode *objlist, Object *obj): give an object into a list of object

4.2.3 Type and Constant

- TypeClass can be int, char or array.
- Type structure includes type class, array size and elementType (if it is an array)
- ConstantValue structure includes type class and value (int or char)
- Type *makeIntType( ): create Integer type
- Type *makeCharType( ): create char type
- Type *makeArrayType(int arraySize, Type *elementType): create array type
- Type *duplicateType(Type *type): create a duplicate type
- int compareType(Type *type1, Type *type2): compare two type
- void freeType(Type 'type): release a type
- ConstantValue *makelntConstant(int I): create constant integer
- ConstantValue *makeCharConstant(int I): create constant char
- ConstantValue *duplicateConstantValue(ConstantValue *v): copy value of constant

4.3 Semantic Rule

4.3.1 Check Identifier

Void checkFreshlident(char *name): check this identifier existing in current

4.3.2 Check declaration of Identifier

Check range of identifier: Ident, Constant, Type, Variable, Function, Procedure, LValueIdent

4.3.3 Check consistency between identifiers defined and identifier using

- Check Int, Char, Array, Basic type
- checkTypeEquality(Type *type1, Type *type2): compare two type