**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY**

\*\*\*\*\*



COMPILER CONSTRUCTION

REPORT

**Student’s name: Phan Quoc Khanh**

**Class: ICT-K56**

**Instructor: Dr. Nguyen Thi Thu Huong**

***Hà Nội, December 2014***

**TABLE OF CONTENTS**

**CHAPTER 1: AN OVERVIEW OF COMPILER**

**CHAPTER 2: LEXICAL ANALYSER FOR KPL**

**CHAPTER 3: SYNTACTIC ANALYSER FOR KPL**

**CHAPTER 4: SEMANTIC ANALYSER FOR KPL**

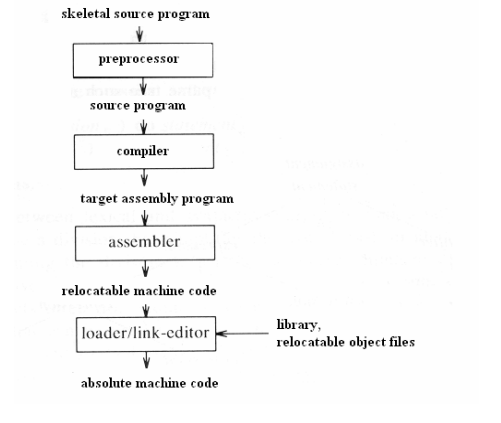
**CHAPTER 1: AN OVERVIEW OF COMPILER**

* 1. **TASK OF A COMPILER**

In simple words, A compiler is a [computer program](http://en.wikipedia.org/wiki/Computer_program) (or set of programs) that transforms [source code](http://en.wikipedia.org/wiki/Source_code) written in a [programming language](http://en.wikipedia.org/wiki/Programming_language) (the source language) into another computer language (the target language, often having a binary form known as [object code](http://en.wikipedia.org/wiki/Object_code)). The most common reason for converting a source code is to create an [executable](http://en.wikipedia.org/wiki/Executable) program. Another critical goal of a compiler is to report errors in source code to developers.

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

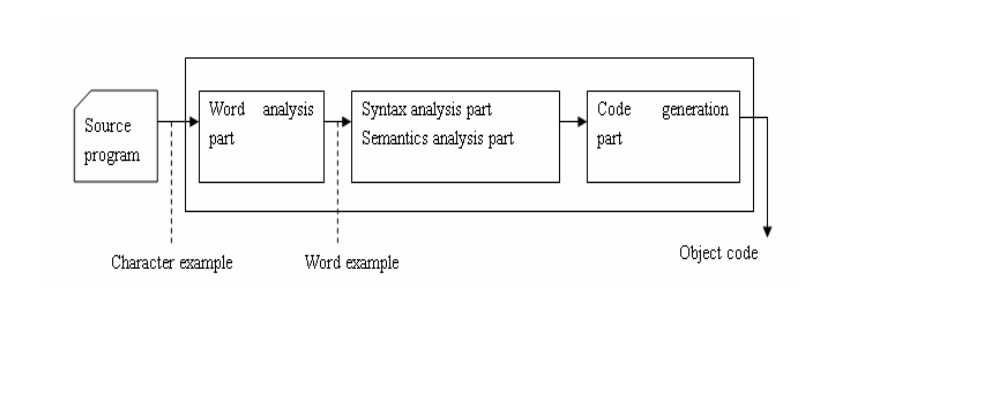
Context of a compiler in a language processor

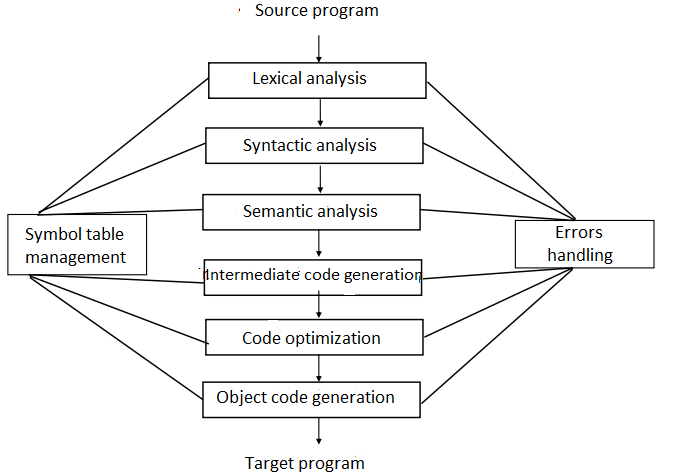


* 1. **COMPONENTS OF A COMPILER**

A typical compiler can be divided into 4 main parts:

* Lexical analyzer
* Syntax analyzer
* Semantic analyzer
* Code generator

****

* 1. **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.
* A typical decomposition is illustrated in this picture:****

Consider an example in KPL:

Sum := initial + increment \* 50 (1)

* + 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, though "scanner" is also used for the first stage of a lexer.

A token is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called tokenization. The characters that form a token are called a lexeme.

(see more at http://en.wikipedia.org/wiki/Lexical\_analysis)

The process of lexical analysis will occur as follows: the scanner will read character-by-character input stream to generate tokens. So the result of the example (1) will be:

1.Identifier (sum).

2.Symbol that represents assignment ( :=)

3.Identifier (initial).

4.Symbol that represents addition (+)

5.Identifier (increment).

6. Symbol that represents multiplication (\*)

7.Number (50)

Note that in the process of lexical analysis, the space, tabulator, and the comments (in KPL // or /\*\*/) will be neglected.

* + 1. **Syntactic analysis**

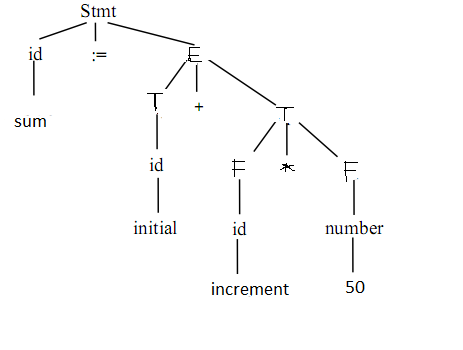
Syntactic analysis (also called parsing) is the process of analysing a [string](http://en.wikipedia.org/wiki/String_(computer_science)) of tokens, conforming to the rules of a [formal grammar](http://en.wikipedia.org/wiki/Formal_grammar) or not. The program that performs parsing is called the syntactic analyser or simply parser.

The output of parsing is the parse tree, or error. Parsing is based on grammar provided to build parse tree.

The most important part of building a compiler is the task of building a grammar that generates structure of a program and cannot be ambiguous. An ambiguous grammar will produce more than one parse tree, therefore must be forbidden.

The above example will produce following parse tree:

(sum := inital + increment \* 50)



* + 1. **Semantic analysis**

Semantic analysis is the phase in which the compiler adds semantic information to the [parse tree](http://en.wikipedia.org/wiki/Parse_tree) and builds the symbol table. This phase performs semantic checks such as [type checking](http://en.wikipedia.org/wiki/Type_checking) (checking for type errors), or [object binding](http://en.wikipedia.org/wiki/Object_binding) (associating variable and function references with their definitions), or [definite assignment](http://en.wikipedia.org/wiki/Definite_assignment_analysis) (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings.

Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the [parsing](http://en.wikipedia.org/wiki/Parsing) phase, and logically precedes the [code generation](http://en.wikipedia.org/wiki/Code_generation_(compiler)) phase.

An important part of semantic analysis is type checking and variable scope checking. In this step, compiler will check, according to specification from source language, if operands of a operation is valid. For example, some programming language require compiler to report error whenever a floating point number is used for indexes of an array. However, some programming language allow type casting, e.g a binary operator can be applied between an integer and a floating point number. In additional, semantic analyser will use symbol table to store information about every identifier, which will generate information about the position of storing identifiers,type and scope of them in program or procedure that include them, or if an identifier is name of function or procedure, it will store information about number and type of parameters, return type, etc.

* + 1. **Intermediate code generation**

After the phase of semantic analysis, some compiler will generate an intermediate representation of source program, known as intermedia code. We can consider this representation as a program for an abstract virtual machine. They have two important properties: easy to generate, and easy to translate into object code. Moreover, intermediate code is machine-independent.

Usually, compiler use three-address codes. Three-address codes are codes that accept at most three parameters, one operator (except assignment). So before generation of these codes, compiler need rules of operator precedence, e.g \* before +.

Example (1) gives the following intermediate code:

T1: = 50;

T2:= id3 \* t1;

T3 := id2 + t2;

Id1:= t3;

* + 1. **Code optimization**

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

For example, the example above (1) can be optimized as:

T1 := id3 \* 50

Id1 := id2 + t1;

There is a significant difference between the amount of optimization codes done by different compiler. In some compiler called “optimize-focused compiler”, a conspicuous proportion of time devoted to this phase. However, there are also some optimization method that can decrese lots of execution time of source program without wasting too much time compiling.

* + 1. **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.

There are a number of factors that affect the design of an object code generator such as: memory management, resource allocation and the sequence of code execution.

* 1. **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 to target program in object codes. This chapter has presented an overview of a compiler in general, including lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, code optimization and object code generation. Output of the preceding phases are always input of the following phases, i.e, output of a scanner (tokens) will be input of the parser, output of the parser (parse tree) will be input of the semantic analyzer, etc.

**CHAPTER 2: DESIGN A LEXICAL ANALYZER FOR KPL**

In a compiler, the program that perform lexical analysis is called the scanner.

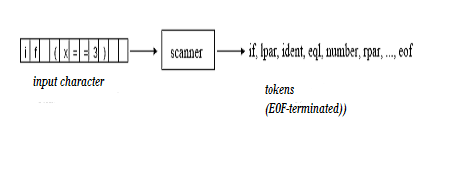
**2.1.TASK OF A SCANNER.**

-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 name, constant names, type names, function names, procedure names:

* 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)

(. và .) to specify indexes in arrays

//, (\*, \*) to indicate comments

-*Others: integer number, string literals…*

**2.3.DATA STRUCTURE IN 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;

Data structure 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;

Stores 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;

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 KPL**

**2.4.1 . Details about functions**

**void** *skipBlank*() : skip spaces.

**void** *skipComment*() : skip comments.

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.

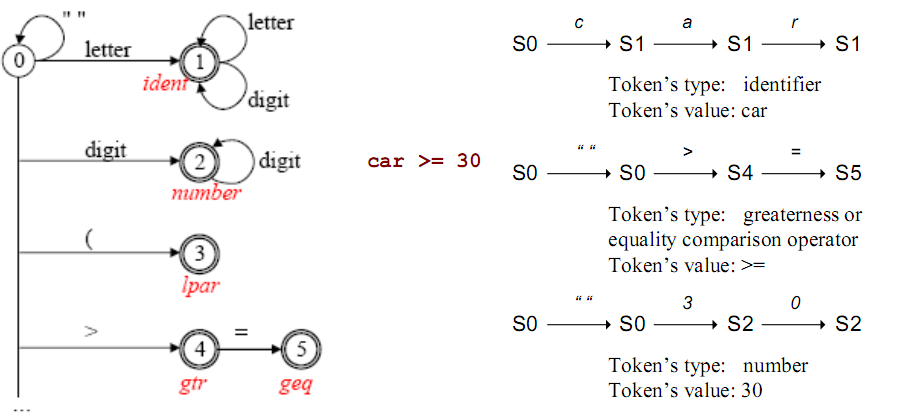
Token\* *makeToken*(TokenType tokenType, int lineNo, int colNo) : create a pointer to a token with predefined type and position.

Token\* *getToken*() : read and return a token (can be invalid token: TOKEN\_NONE).

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

**2.4.2.Details about execution of a scanner.**

Scanner is a finite automation. Everytime it return a token, the state will be 0. When detects invalid characters, the state will be -1.



During the reading of input stream, *getToken*() will be looped until meet (EOF).

- Detect spaces (CHAR\_SPACE) -> *skipBlank*(), state will be 0, -> *getToken*()…

- Every tokens read will be passed to parser for the next phase of compiling process.

Details about implementation of functions, refer to file [scanner.c](scanner/scanner.c)

**CHAPTER 3: DESIGN SYNTACTIC ANALYSER FOR KPL**

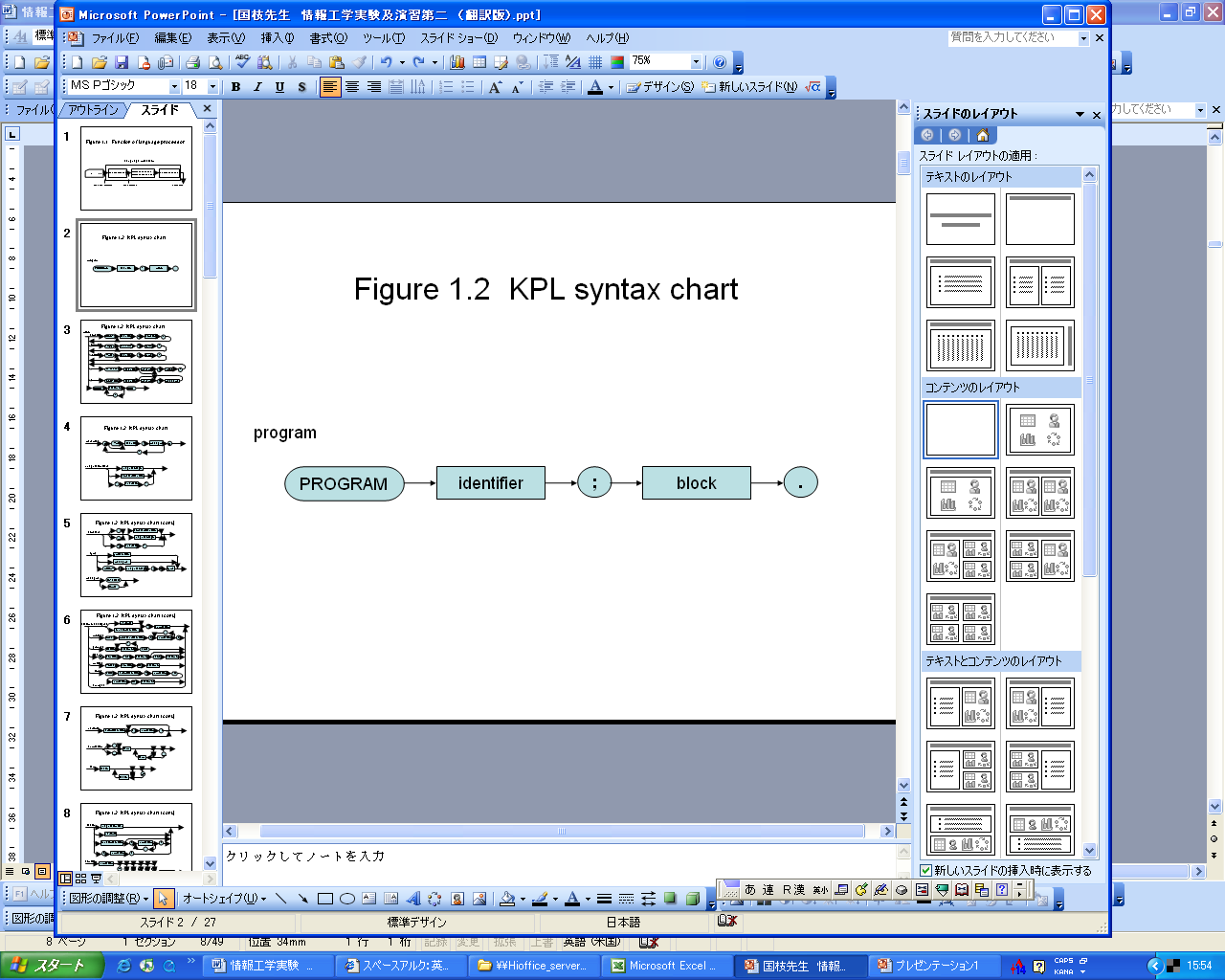
**3.1. Task of a syntatic analyser (parser)**

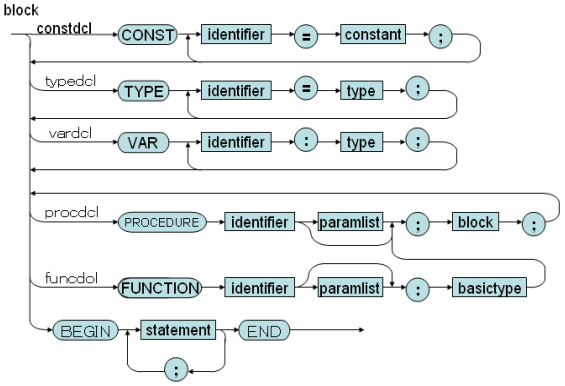
- Check the syntax of the program for errors.

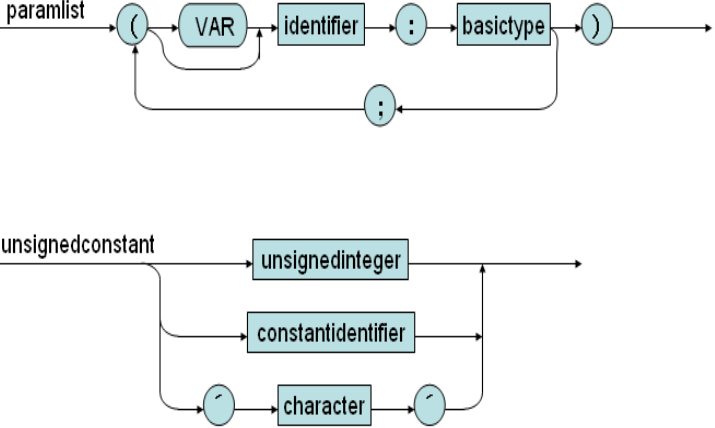
- Produce parse tree for semantic analyser otherwise.

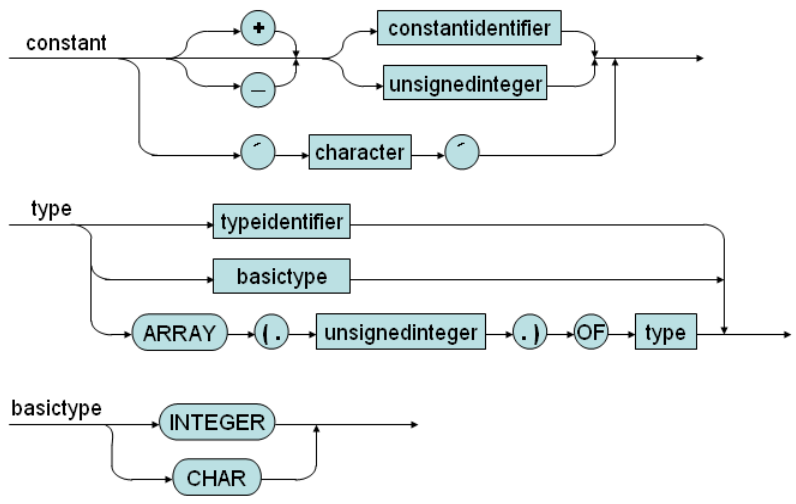
**3.2. Syntax diagram and BNF grammar**

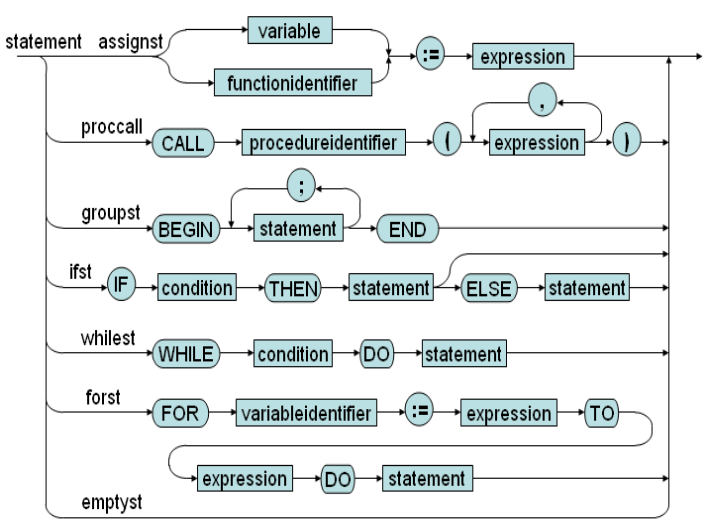
**3.2.1. Syntax diagram.**

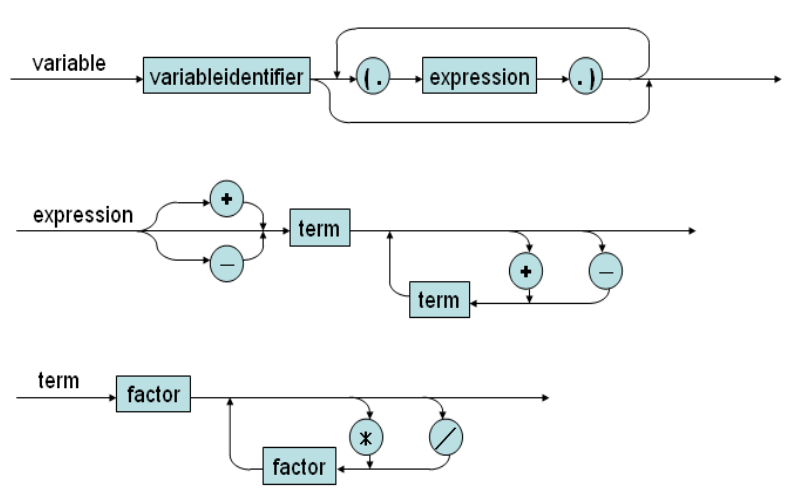


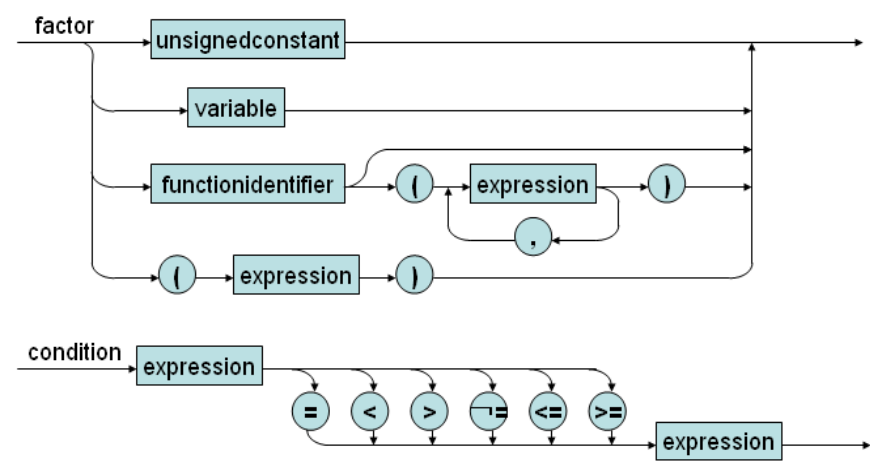


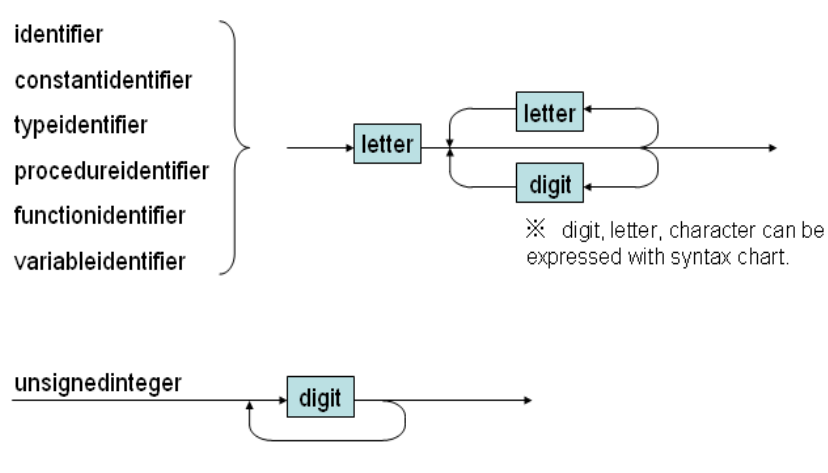












**3.2.1. BNF grammar.**

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

10) ConstDecls::= ConstDecl ConstDecls

11) ConstDecls::= Ɛ

12) ConstDecl ::= Ident SB\_EQUAL Constant SB\_SEMICOLON

13) TypeDecls ::= TypeDecl TypeDecls

14) TypeDecls ::= Ɛ

15) TypeDecl ::= Ident SB\_EQUAL Type SB\_SEMICOLON

16) VarDecls ::= VarDecl VarDecls

17) VarDecls ::= Ɛ

18) VarDecl ::= Ident SB\_COLON Type SB\_SEMICOLON

19) SubDecls ::= FunDecl SubDecls

20) SubDecls ::= ProcDecl SubDecls

21) SubDecls ::= Ɛ

22) FunDecl ::= KW\_FUNCTION Ident Params SB\_COLON BasicType

SB\_SEMICOLON Block SB\_SEMICOLON

23) ProcDecl ::= KW\_PROCEDURE Ident Params SB\_SEMICOLON Block

SB\_SEMICOLON

24) Params ::= SB\_LPAR Param Params2 SB\_RPAR

25) Params ::= Ɛ

26) Params2 ::= SB\_SEMICOLON Param Params2

27) Params2 ::= Ɛ

28) Param ::= Ident SB\_COLON BasicType

29) Param ::= KW\_VAR Ident SB\_COLON BasicType

30) Type ::= KW\_INTEGER

31) Type ::= KW\_CHAR

32) Type ::= TypeIdent

33) Type ::= KW\_ARRAY SB\_LSEL Number SB\_RSEL KW\_OF Type

34) BasicType ::= KW\_INTEGER

35) BasicType ::= KW\_CHAR

36) UnsignedConstant ::= Number

37) UnsignedConstant ::= ConstIdent

38) UnsignedConstant ::= ConstChar

40) Constant ::= SB\_PLUS Constant2

41) Constant ::= SB\_MINUS Constant2

42) Constant ::= Constant2

43) Constant ::= ConstChar

44) Constant2::= ConstIdent

45) Constant2::= Number

46) Statements ::= Statement Statements2

47) Statements2 ::= KW\_SEMICOLON Statement Statement2

48) Statements2 ::= Ɛ

49) Statement ::= AssignSt

50) Statement ::= CallSt

51) Statement ::= GroupSt

52) Statement ::= IfSt

53) Statement ::= WhileSt

54) Statement ::= ForSt

55) Statement ::= Ɛ

56) AssignSt ::= Variable SB\_ASSIGN Expession

57) AssignSt ::= FunctionIdent SB\_ASSIGN Expression

58) CallSt ::= KW\_CALL ProcedureIdent Arguments

59) GroupSt ::= KW\_BEGIN Statements KW\_END

60) IfSt ::= KW\_IF Condition KW\_THEN Statement ElseSt

61) ElseSt ::= KW\_ELSE statement

62) ElseSt ::= Ɛ

63) WhileSt ::= KW\_WHILE Condition KW\_DO Statement

64) ForSt ::= KW\_FOR VariableIdent SB\_ASSIGN Expression KW\_TO

Expression KW\_DO Statement

65) Arguments ::= SB\_LPAR Expression Arguments2 SB\_RLAR

66) Arguments ::= Ɛ

67) Arguments2::= SB\_COMMA Expression Arguments2

68) Arguments2::= Ɛ

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

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 ::= Ɛ

82) Term ::= Factor Term2

83) Term2 ::= SB\_TIMES Factor Term2

84) Term2 ::= SB\_SLASH Factor Term2

85) Term2 ::= Ɛ

86) Factor ::= UnsignedConstant

87) Factor ::= Variable

88) Factor ::= FunctionApptication

89) Factor ::= SB\_LPAR Expression SB\_RPAR

90) Variable ::= VariableIdent Indexes

91) FunctionApplication ::= FunctionIdent Arguments

92) Indexes ::= SB\_LSEL Expression SB\_RSEL Indexes

93) Indexes ::= Ɛ

**3.3. Recursive descent parsing**

*-Properties:*

+ LL(k) is the language that needs looking 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 (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.

**3.4. Data structure in parser for KPL**

Use data structure in scanner.

**3.5. Parse 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.

**3.6. Parsing 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](parser/parser.c)

# CHAPTER 4. DESIGN SEMANTIC ANALYZER FOR KPL

* 1. **Tasks of semantic analyzer**

Import tasks of a sematic analyser:

- Produce symbol table for future references (eg. scope and type checking).

-Scope checking

-Type checking.

* 1. **Design symbol table**
     1. **Why we need symbol table**

We need a symbol table to store information needed about every identifiers in the program.

Each [identifier](http://en.wikipedia.org/wiki/Identifier) in a program's [source code](http://en.wikipedia.org/wiki/Source_code) is associated with information relating to its declaration or appearance in the source, such as its [type](http://en.wikipedia.org/wiki/Data_type), [scope](http://en.wikipedia.org/wiki/Scope_(programming)) level and its [location](http://en.wikipedia.org/wiki/Memory_address).

* + 1. **Design symbol table**

**4.2.2.1.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.

enum ObjectKind {

OBJ\_CONSTANT,

OBJ\_VARIABLE,

OBJ\_TYPE,

OBJ\_FUNCTION,

OBJ\_PROCEDURE,

OBJ\_PARAMETER,

OBJ\_PROGRAM

};

Kinds of object (a variable object, a constant object, etc)

enum ParamKind {

PARAM\_VALUE,

PARAM\_REFERENCE

};

Kinds of parameter: value or reference type.

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.

struct ObjectNode\_ {

Object \*object;

struct ObjectNode\_ \*next;

};

A linked list to represent list of objects.

struct ConstantValue\_ {

enum TypeClass type;

union {

int intValue;

char charValue;

};

};

typedef struct ConstantValue\_ ConstantValue;

struct Scope\_;

struct ObjectNode\_;

struct Object\_;

struct ConstantAttributes\_ {

ConstantValue\* value;

};

struct VariableAttributes\_ {

Type \*type;

struct Scope\_ \*scope;

};

struct TypeAttributes\_ {

Type \*actualType;

};

struct ProcedureAttributes\_ {

struct ObjectNode\_ \*paramList;

struct Scope\_\* scope;

};

struct FunctionAttributes\_ {

struct ObjectNode\_ \*paramList;

Type\* returnType;

struct Scope\_ \*scope;

};

struct ProgramAttributes\_ {

struct Scope\_ \*scope;

};

struct ParameterAttributes\_ {

enum ParamKind kind;

Type\* type;

struct Object\_ \*function;

};

typedef struct ConstantAttributes\_ ConstantAttributes;

typedef struct TypeAttributes\_ TypeAttributes;

typedef struct VariableAttributes\_ VariableAttributes;

typedef struct FunctionAttributes\_ FunctionAttributes;

typedef struct ProcedureAttributes\_ ProcedureAttributes;

typedef struct ProgramAttributes\_ ProgramAttributes;

typedef struct ParameterAttributes\_ ParameterAttributes;

To store typical attributes of each type: e.g a constant type must have a constant value, a function type must have a parameter list, a return type, and own a scope.

**4.2.2.2.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.

* 1. **Verify scoping rules**
     1. **Checking fresh identifier**

We determine if an identifier is not declared yet, by function void checkFreshIdent(char \*name)

void checkFreshIdent(char \*name) {

if (findObject(symtab->currentScope->objList, name) != NULL)

error(ERR\_DUPLICATE\_IDENT, currentToken->lineNo, currentToken->colNo);

}

If the identifier is already declared, findObject function will return an non-null value.

* + 1. **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.

* 1. **Type checking**
     1. **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.

**4.4.2. Checking specific requirements in some statement (e.g. LValue in assign statement.**