**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY**

\*\*\*\*\*



COMPILER CONSTRUCTION

REPORT

**Student’s name: Le Van Vuong (20154420)**

**Class: ICT-K60**

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

***Hà Nội, January 2019***

**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**

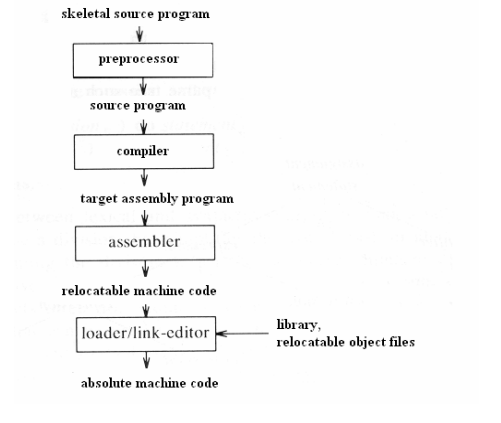
**CHAPTER 1: AN OVERVIEW OF COMPILER**

* 1. **TASK OF A COMPILER**

In simple words, A compiler is a [computer program](http://en.wikipedia.org/wiki/Computer_program" \o "Computer program) (or set of programs) that transforms [source code](http://en.wikipedia.org/wiki/Source_code" \o "Source code) written in a [programming language](http://en.wikipedia.org/wiki/Programming_language" \o "Programming language) (the source language) into another computer language ([object code](http://en.wikipedia.org/wiki/Object_code" \o "Object code)). The most common reason for converting a source code is to create an [executable](http://en.wikipedia.org/wiki/Executable" \o "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>)

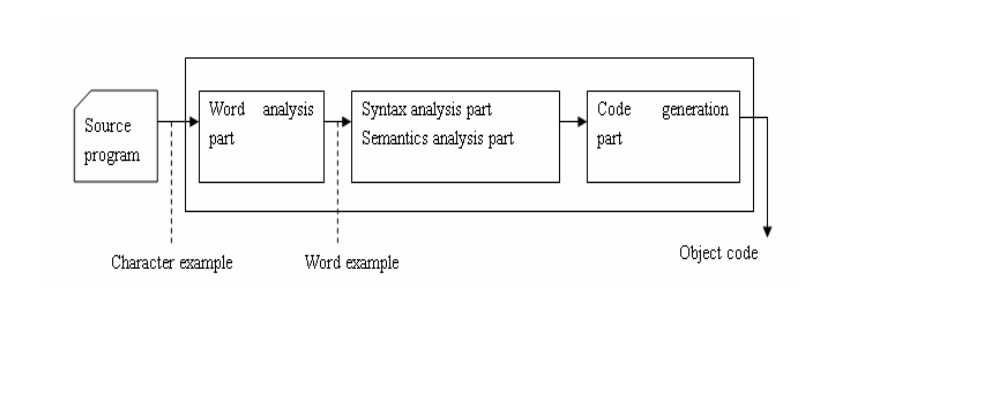
Context of a compiler in a language processor

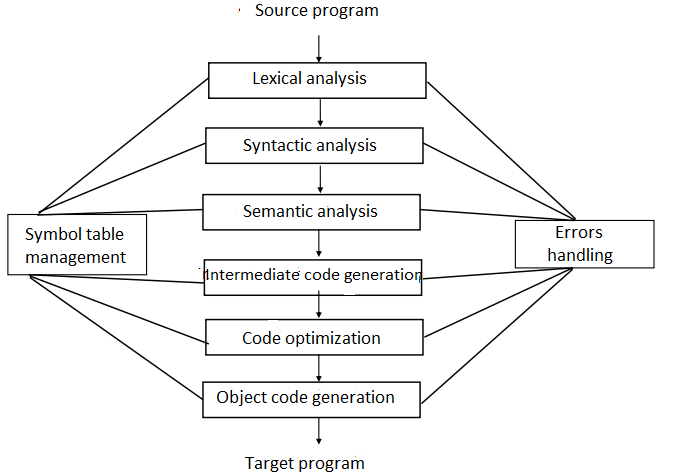


* 1. **COMPONENTS OF A COMPILER**

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 (Scanner)**

Stream of characters making up the source program is read

from left to right and grouped into tokens.

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)

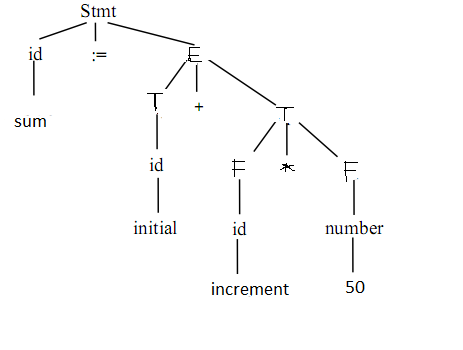
Note that in the process of lexical analysis, the space, tabulator, and the comments (in KPL // or /\*\*/) will be neglected.

* + 1. **Syntactic analysis (Parser)**

Group the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. 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 above example will produce following parse tree:



* + 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" \o "Parse tree) and builds the symbol table. This phase performs semantic checks such as [type checking](http://en.wikipedia.org/wiki/Type_checking" \o "Type checking) (checking for type errors), or [object binding](http://en.wikipedia.org/wiki/Object_binding" \o "Object binding) (associating variable and function references with their definitions), or [definite assignment](http://en.wikipedia.org/wiki/Definite_assignment_analysis" \o "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" \o "Parsing) phase, and logically precedes the [code generation](http://en.wikipedia.org/wiki/Code_generation_(compiler)" \o "Code generation (compiler)) phase.

An important part of semantic analysis is type checking the source program for semantic errors and gather type information for the subsequent code generation phase.

* + 1. **Intermediate code generation**

After the phase of semantic analysis, some compiler will generate an intermediate representation of source program, known as intermediate 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.

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 decrease 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.

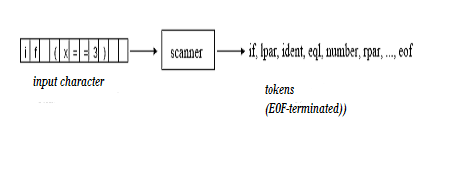
**2.1.TASK OF A SCANNER.**

- 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**

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;

**typedef struct** {

char string[MAX\_IDENT\_LEN + 1]; // content of token

int lineNo, colNo; // position of token

TokenType tokenType; // type of token

int value; // value of token if a number.

} Token;

**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.

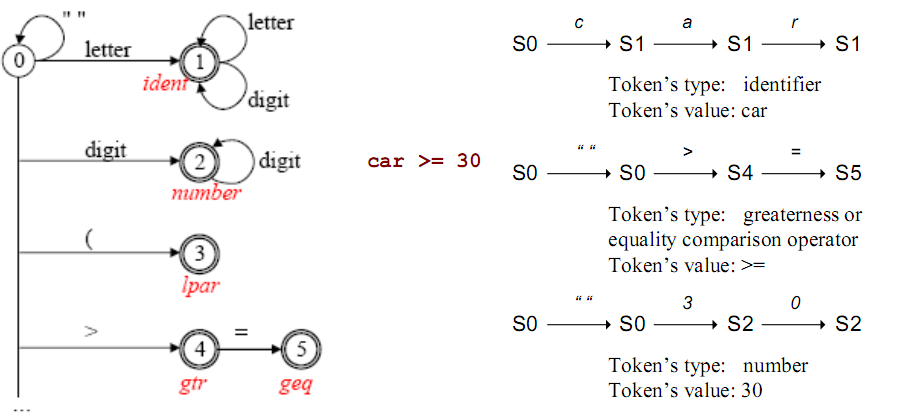
Token\* *readNumber*() : *read a integer number, return a pointer of Token type.*

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.

**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**

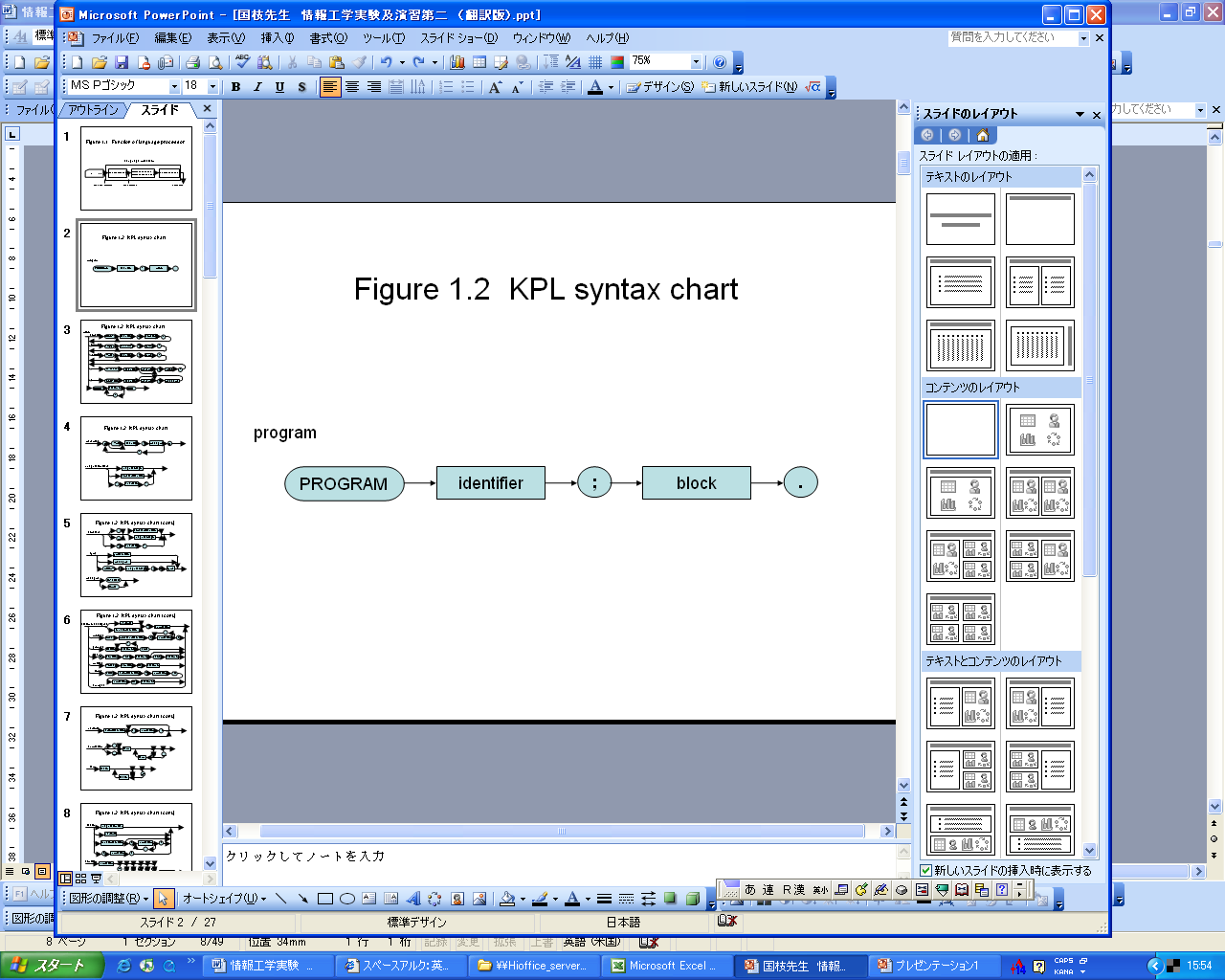
**3.1. Task of a syntatic analyser (parser)**

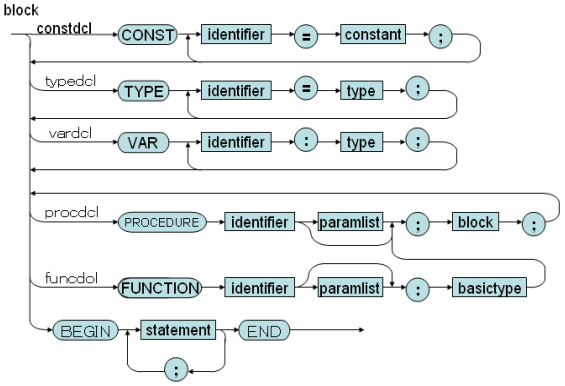
- Check the syntax of the program for errors.

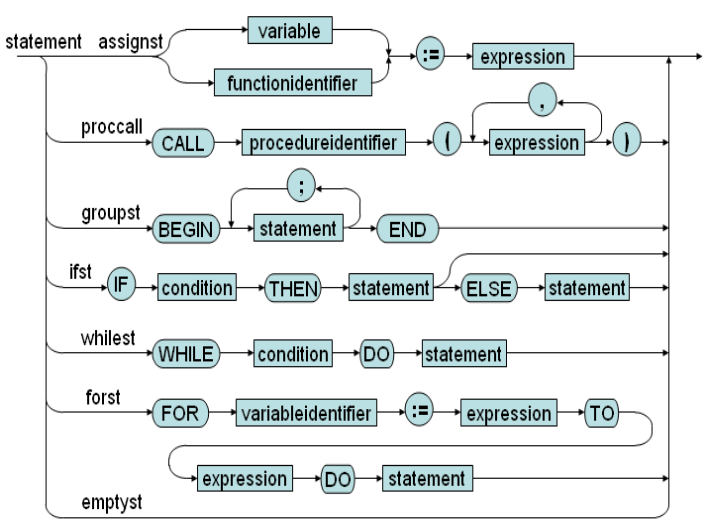
- Produce parse tree for semantic analyser otherwise.

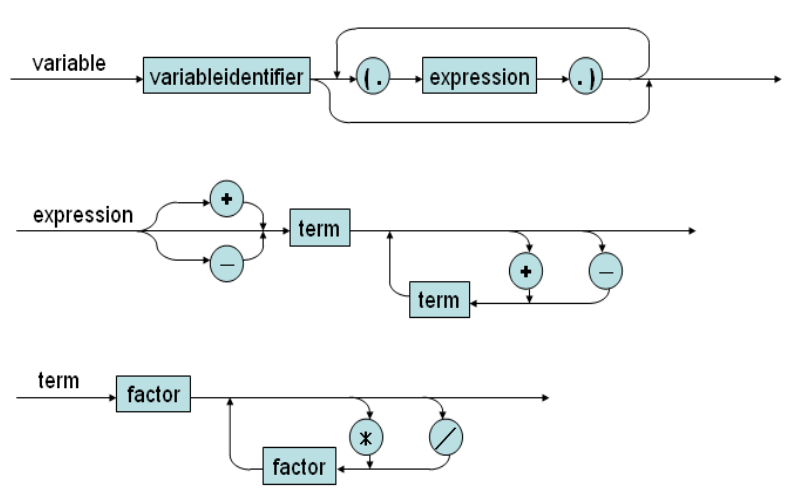
**3.2. Syntax diagram and BNF grammar**

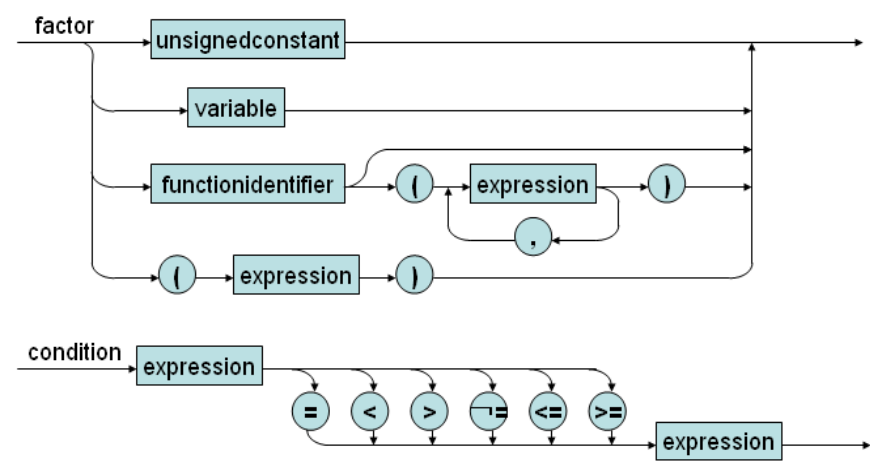
**3.2.1. Syntax diagram.**











**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

...

37) UnsignedConstant ::= ConstIdent

38) UnsignedConstant ::= ConstChar

...

47) Statements2 ::= KW\_SEMICOLON Statement Statement2

...

49) Statement ::= AssignSt

50) Statement ::= CallSt

51) Statement ::= GroupSt

52) Statement ::= IfSt

53) Statement ::= WhileSt

54) Statement ::= ForSt

...

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

...

1. Condition ::= Expression Condition2
2. ...

75) Expression ::= SB\_PLUS Expression2

76) Expression ::= SB\_MINUS Expression2

...

1. Term ::= Factor Term2

...

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 non-terminal 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** *compileConstDecls*(void): parse constant declarations.

**void** *compileConstDecl*(void): parse a single constant declaration.

...

**void** *compileFuncDecl*(void): parse function declarations.

**void** *compileStatements*(void): parse all statements.

**void** *compileIfSt*(void): parse an IF statement.

...

**void** *compileArguments*(void): parse list of arguments passed to a function or procedure.

**void** *compileTerm*(void): compile a term, which can be composed of (\*, /) of compileFactor

**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" \o "Identifier) in a program's [source code](http://en.wikipedia.org/wiki/Source_code" \o "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" \o "Data type), [scope](http://en.wikipedia.org/wiki/Scope_(programming)" \o "Scope (programming)) level and its [location](http://en.wikipedia.org/wiki/Memory_address" \o "Memory address).

* + 1. **Design symbol table**

**4.2.2.1.Data structure in symbol table**

+ The data structure to represent symbol table itself:

struct SymTab\_ {

Object\* program; // the program object.

Scope\* currentScope; // current scope of symbol table

ObjectNode \*globalObjectList; // store global objects such as // functions: CALLI, WRITEI, etc.

};

+ To store information about each object in program, such as main program itself, a procedure or function, a variable, a constant, 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;

};

};

struct Scope\_ {

ObjectNode \*objList;

Object \*owner;

struct Scope\_ \*outer;

};

**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\* createFunctionObject(char \*name): create a function object.

...

Type\* makeIntType(void): create an integer type.

...

Type\* duplicateType(Type\* type): copy type.

int compareType(Type\* type1, Type\* type2): compare type

...

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)

Object\* checkDeclaredConstant(char \*name): check declared constants.

...

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 checkTypeEquality(Type\* type1, Type\* type2): *check for equality of types, if not, report an error message.*