Hanoi university of Science and Technology
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Compiler Constructor Report

Final report for Compiler Constructor Lab course.

By Le Xuan Thanh ICT.02-K61, HUST, January 6th, 2020.

All I have learnt about Compiler

Thank you for reading my report about the Compiler. You can consider this is as a short diary about what my learning process at the best Compiler Constructor in HUST. My academic english is not really well so hopefully forgiveable by the readers. So shall we start!

“

A computer program that translates computer code written in one programming language into machine language

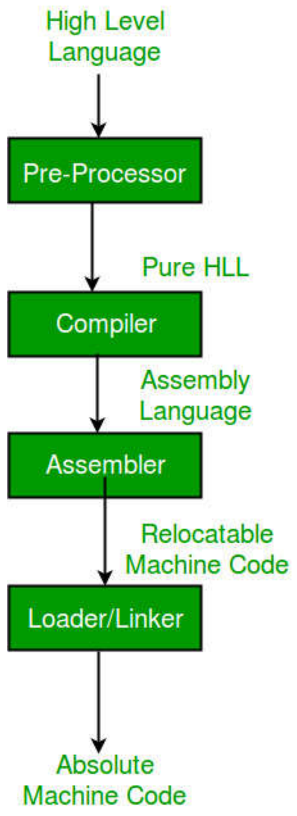
”

If you are the developer, you *MUST BE* using the compiler everyday! Believe or not.

Compiler is a magical tool the help convert your code into series of “1” and “0” which known as *Machine Code*. GCC or G++ are famous compiler of C as an example. Thanks to the development of technology alot of new programming language came out; but all it is, they are just simple creating new compiler! Why? From what I mentioned, compiler are just **set of conventions of language.** The easiest example, just looking at the way you implementing your functions and procedures. The meaning of them are the same. They are just changing due to the specification among the languages. Imagine of the world without compiler? Human now have to type 101000... to communicate with the machine. May be it is impossible! :D

Compilers make developers’ life way more peaceful and colorful!

What a compiler really is?



Take the most famous famouse compiler like GCC+ as an example. It will do for us some serveral steps, integrated at one to make a fully working compiler like: Pre-Process, Compile, Assemble and Loader/Link. However, in this scope of the course, we are just need to focus on the really part of the compiler

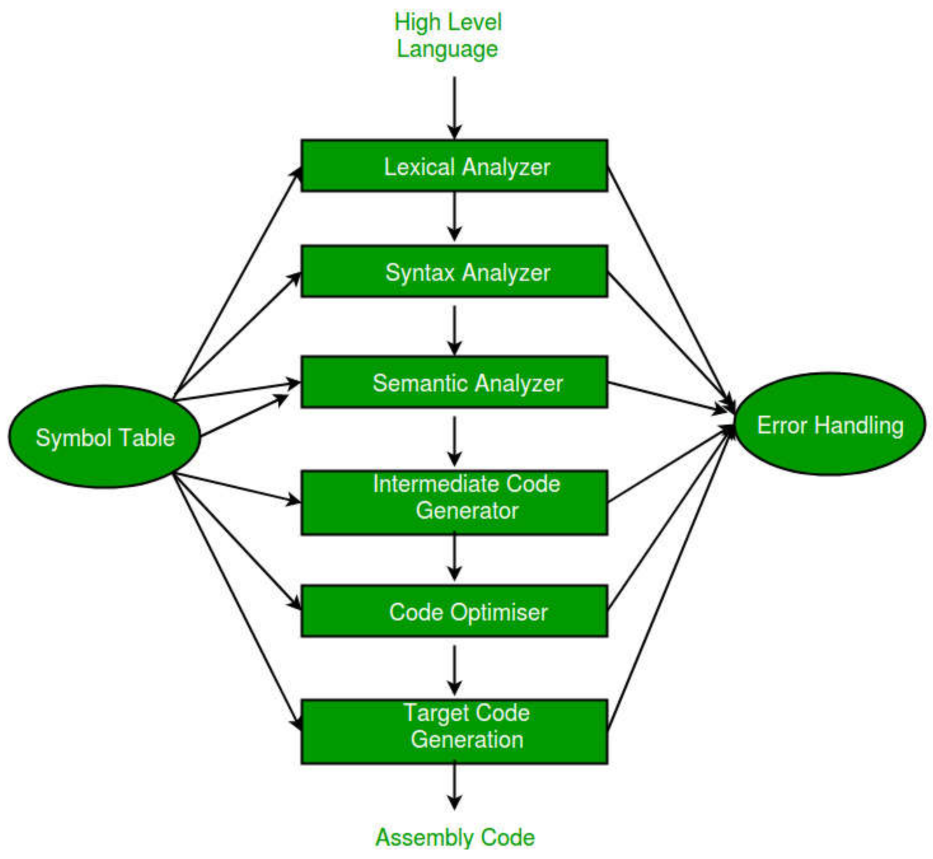
Which is the second step in the figure on the right.

Compilers insight!

Now let us dive deeper and see how it has been created.

Here is the core part of the compilier, where the logic of converting a High level language into Assembly Code. This is the truely part of every compiler. However in this report I will just cover three of them as this is what we have learnt about:

- Lexical Analyzer: convert HLL to stream of tokens. Also called as Scanner.
- Syntax Analyzer: convert tokens into syntax tree. As known as Parser.
- Semantic Analyzer: uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition.



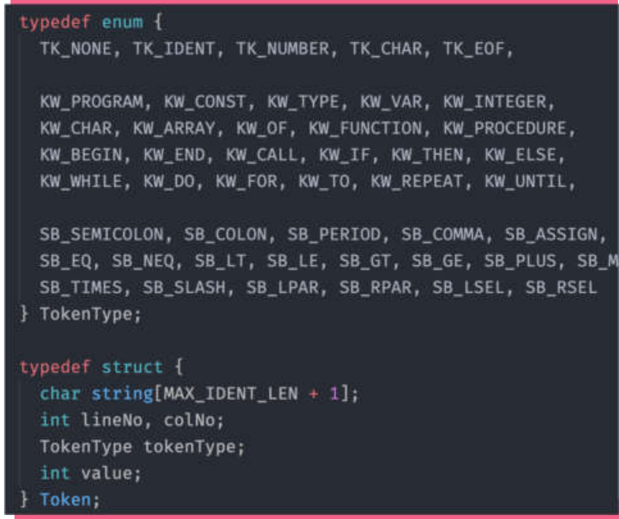
Lexical Analyzer

Lexical Analyzer (Scanner) converts the stream of input characters into a stream of tokens, i.e: meaningful character strings. The process of lexical analyzer will occur as follows: the scanner will read character - by - character input stream to generate tokens.

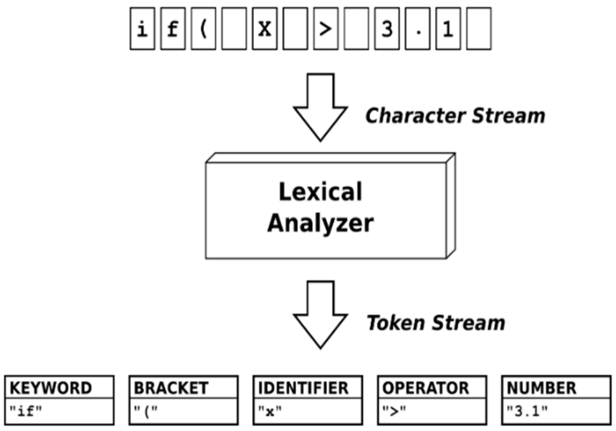
Lexical Analyzer Tasks

There are three main tasks which scanner is responsible for:

- Neglect meaningless character: space, tabulalor, EOF, CR, LF, comments.
- Detect invalid symbols: @, ! (stand-alone), etc. Whenever the lexical analyzer finds an invalid token, it generates an error.
- Recognize different types of token:
- Pass recognized tokens to the parser to perform job of syntactic analysis.

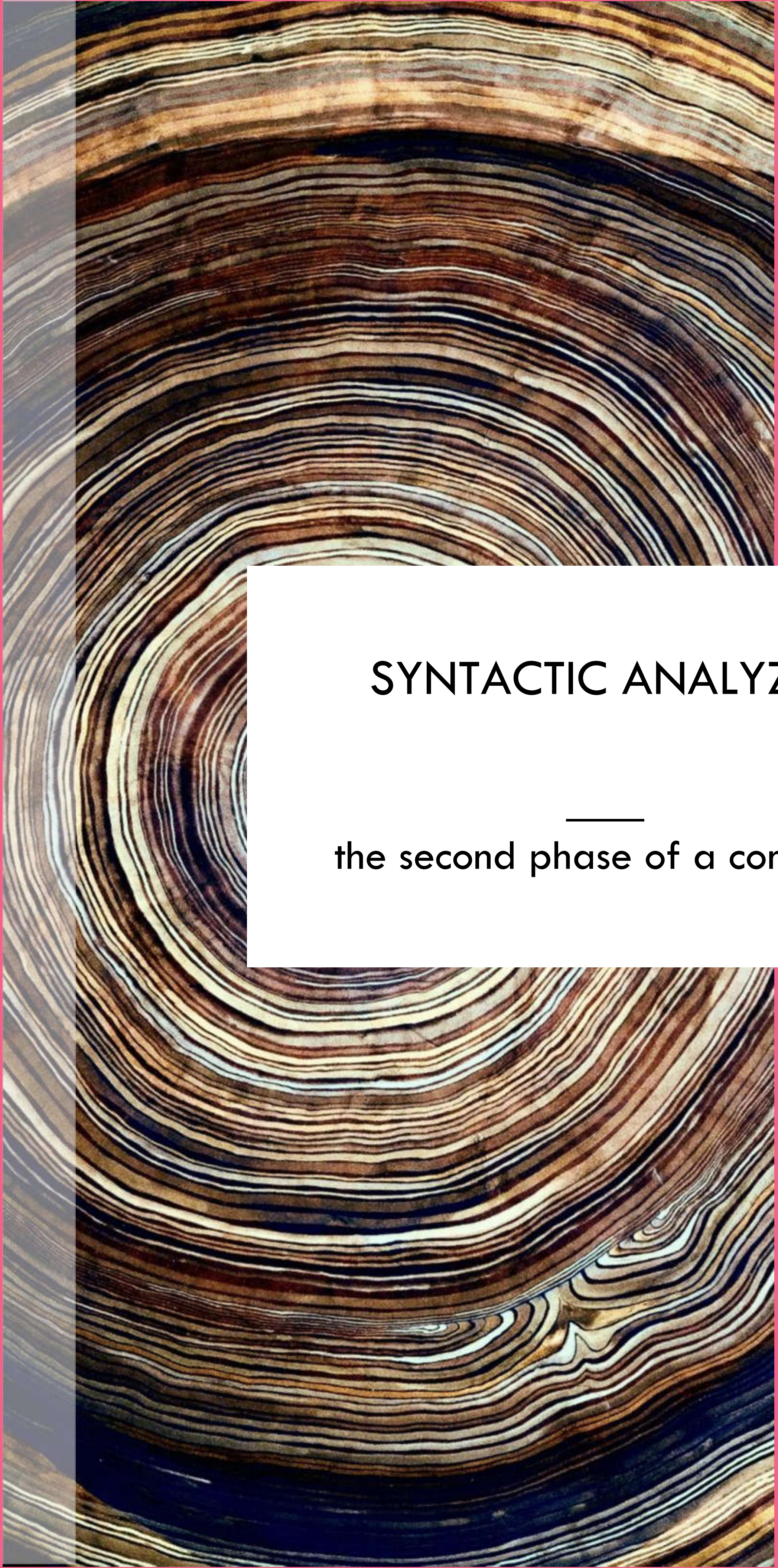


```c
typedef enum {
  TK_NONE, TK_IDENT, TK_NUMBER, TK_CHAR, TK_EOF,

  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, KW_REPEAT, KW_UNTIL,

  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_M
  SB_TIMES, SB_SLASH, SB_LPAR, SB_RPAR, SB_LSEL, SB_RSEL
} TokenType;

typedef struct {
  char string[MAX_IDENT_LEN + 1];
  int lineNo, colNo;
  TokenType tokenType;
  int value;
} Token;
```



Main functions

- ✓ *void skipBlank()*: skip characters: blank character, tab, new line.
- ✓ *void skipComment()*: skip comment (string that start by *(** and end by **)*)
- ✓ *Token *readIdentKeyword(void)*: if the first character of string is alphabet, it can be a identifier or keyword.
- ✓ *Token *readNumber(void)*: if the first character of string is number, it is a number.
- ✓ *Token *readConstChar(void)*: if the first character of string is single quote, it is a character
- ✓ *Token *getToken(void)*: read characters from source code and use these functions to create a new token.



SYNTACTIC ANALYZER

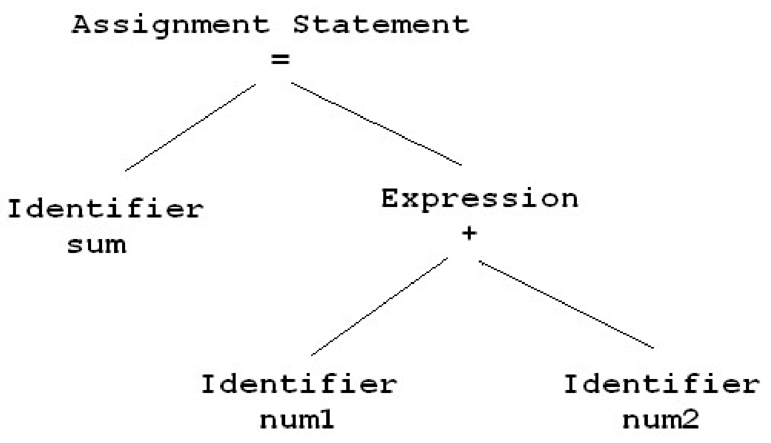
the second phase of a compiler

Syntatic Analyzer tasks

The tasks of a syntatic Analyzer:

- ✓ Take the input from a lexical analyzer in form of token streams.
- ✓ Check the syntax of the program for errors.
- ✓ Produce parse tree for semantic analyzer.

A parse tree is a power tool to define who algorithm execute. The logic of reading the token. I found that this is very interesting as this is the commucation tool between High level language and lower ones. This is an example of **Assignment Statement** below in form of *BNF Grammar*



Token *currentToken;

Token *lookAhead;

These are two things I really like inside Syntatic Analyzer. It reminds me of linked-list data structure. A very beautiful data structure.

The below images from now on is taken from our C project on laboratory.

There are serveral way of designing a syntax as a top-down parsing:

- Token LookAhead.
- Parsing terminal symbol.
- Parsing non-terminal symbol.

The way of designing this varies however still aiming to have a tree "*unambigious*" though. This means: there should be only one output maded for every different inputs. The result of them should be unique.

Main functions

---

1. Scan function use for changing *currentToken* to *lookAhead* and get new *lookAhead* token.

```
void scan(void) {
  Token* tmp = currentToken;
  currentToken = lookAhead;
  lookAhead = getValidToken();
  free(tmp);
}
```

2. And ofcourse, we need a function to “eat” the current token based on the *tokenType* and move to the next one.

```
void eat(TokenType tokenType) {
  if (lookAhead->tokenType == tokenType) {
    printToken(lookAhead);
    scan();
  } else missingToken(tokenType, lookAhead->lineNo,
lookAhead->colNo);
}
```

Design:

SEMANTIC ANALYZER

Semantic analyzer is the third phase of a compiler, it 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.

Tasks of Semantic analyzers

Semantic helps interpret symbols, types and the relationship with each other. It used to check if the syntax structure constructed of a program derived meaning or not.

It contains 3 tasks:

1. Produces symbol table for future references
2. Scope checking.
3. Type checking.

Data Structure of Semantic Analyzer

Symbol Table structure to store the whole Symbol Table of the running program.

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

We need a symbol table to:

- ✓ Store information about each object in program such as main program, procedure, function, variable, constant,…
- ✓ Store typical attributes for each type: a function must have a parameter list and return type

Data Structure of Object

```
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;
  };
};
```

Main functions

To check type or check kind of Object, we need to find it first.

```
Scope* scope = symtab->currentScope;
  Object* obj;

  while (scope != NULL) {
    obj = findObject(scope->objList, name);
    if (obj != NULL) return obj;
    scope = scope->outer;
  }
  obj = findObject(symtab->globalObjectList, name);
  if (obj != NULL) return obj;
  return NULL;
}
```

Here is the list of the functions in order to check the Declaration Step:

```
void checkFreshIdent(char *name);
Object* checkDeclaredIdent(char *name);
Object* checkDeclaredConstant(char *name);
Object* checkDeclaredType(char *name);
Object* checkDeclaredVariable(char *name);
Object* checkDeclaredFunction(char *name);
Object* checkDeclaredProcedure(char *name);
Object* checkDeclaredLValueIdent(char *name);
```

List functions for checking type:

```
void checkIntType(Type* type);
void checkCharType(Type* type);
void checkArrayType(Type* type);
void checkBasicType(Type* type);
void checkTypeEquality(Type* type1, Type* type2);
```

Type checker verifies that the type of a construct (constant, variable, array, list, object) matches what is expected in its usage context.

REFERENCES

1. **Lecture Slides: Compiler Construction 2019-2020**

   Dr. Nguyễn Thi Thu Hương (HUST)
2. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
3. https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/
4. https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/