

Notes on Compiler Constructions

Nuo Yan

Table of Contents:

1. Introduction
2. Overview of Compilers
3. Scanner
4. Parser and Code Generation
5. Conclusions

Introduction:

This document briefly introduces the concepts and construction of programming language compilers. The document uses a simple programming language we named “D” for the purposes of this project, then goes through the process of constructing a full compiler for the “D” programming language. “D” only represents the name of the sample programming language used in this project, and has no relationship with any popular existing programming language. Code samples will be included in this document; however, since this document will be published on the Internet, to prevent future computer science students with similar assignments from plagiarizing any part of this paper, the full source code for the compiler project will not be included in this document. All source codes for the compiler in this document are written in C, and compiled using `gcc` in Unix/Linux. The source code has also been tested for compilation under Windows NT (including 2000, XP, and Vista) using Visual C++ (including 6.0, 2003, and 2005). The X86 assembly language code presented in this documents uses the Microsoft/ Intel standard, which is different from the GNU/AT&T standard in some respects. This document is neither a course requirement, nor an assignment. All contents represent the author’s personal interests and the author provides no warranty.

Overview of Compilers:

Compilers translate computer programs written in source languages into target languages. For instance, programs written in C/C++ will be translated into an assembly language while the programs are being compiled; programs written in Java will be translated into Java virtual machine language. Then the Java virtual machine language will be compiled and translated into the SPARC or X86 machine code, and programs written in COBOL will be translated into IBM mainframe machine language.

There are two main parts to forming a full compiler: analysis and synthesis. Each part includes several sub-parts.

The analysis includes the following:

1. Lexical analysis:

This part is also called the *scanner*; it converts the source program from a character stream to a token stream, and ignores any comments or white space from the source program. By doing so, the next part (we will call it the *parser*) will only read the tokens that include useful source code information (e.g., keywords, identifiers, numbers, etc.) but not comments or white spaces. In other words, the *scanner* processes the source code, eliminates all comments and white spaces, and provides the remaining useful source code in the form of tokens to the *parser* one at a time.

2. Parser:

It reads the token stream, one at a time, to construct syntax trees. The future parts, including target code generation, should be implemented in the parser to generate the target code while the parser is constructing the syntax trees.

3. Symbol Table:

Different symbol tables that store function names, declared identifiers, parameters, etc. will be used in the compiling process for processing declarations, checking type, etc. Symbol tables can be implemented in many different ways. In this document, they are implemented by using binary search trees.

4. Optimization:

In production compilers, this part includes different code improvement transformations and resource allocation decisions. In this document, the compiler project ignores this part.

5. Code Generation:

This part generates the target language code.

For example, there is a simple program with the following source code:

```
//simple sample program in "D"  
int sample_program(int a, int b)  
{  
    int c; int d;  
    return c + d; //return the sum of c and d  
}
```

The *scanner* will eliminate comments and white spaces, then convert the character stream into a token stream. The following code fragment shows examples of what the token stream of the

previous source code looks like.

```
TOK_INT TOK_ID LPAREN TOK_INT TOK_ID COMMA TOK_INT TOK_ID RPAREN LBRACKET  
TOK_INT TOK_ID SCOLON TOK_INT TOK_ID RETURN TOK_ID PLUS TOK_ID SCOLON RBRACKET
```

In the parsing process, the parser reads the token stream; analyzes the grammar structure; adds the appropriate names of the declaration, parameters, functions, etc. to the symbol tables; and then generates a target code when appropriate, which in this project, is the X86 assembly code. An example of the X86 assembly language code for the previous “D” programming language source code looks like this:

```
sample_program:  
    push    ebp  
    mov     ebp, esp  
    sub     esp, 8  
    mov     eax, [ebp+12]  
    mov     [ebp-4], eax  
    mov     eax, [ebp+8]  
    mov     [ebp-8], eax  
    mov     eax, [ebp-4]  
    push   eax  
    mov     eax, [ebp-8]  
    pop    ecx  
    add    eax, ecx  
    mov     esp, ebp  
    pop    ebp  
    ret
```

Scanner:

The work that the *scanner* should be doing is to take the character stream of the source code as input, and produce the token stream as output. The overall structure of the scanner should contain a method that returns a token when it is called. As a result, the functions defining tokens should be created before starting to work on the scanner itself.

What can be counted as a token? As introduced in the “Overview of Compilers” section of this document, the only two things the *scanner* does not process are comments and white spaces. As a result, the tokens can be defined as follows, in the header file **token.h**:

```

/*----- EOF, ID, and INT -----*/
#define TOK_EOF    0  /* end of file */
#define TOK_ID     1  /* identifier */
#define TOK_INTGR  2  /* integer */
/*----- D keywords -----*/
#define TOK_INT    10 /* int */
#define TOK_IF     11 /* if */
#define TOK_ELSE   12 /* else */
#define TOK_WHILE  13 /* while */
#define TOK_RETURN 14 /* return */
/*----- Operators, punctuation, and delimiters */
#define LPAREN     20 /* ( */
#define RPAREN     21 /* ) */
#define COMMA      22 /* , */
#define BECOMES    23 /* = */
#define EQUALS     24 /* == */
#define PLUS       25 /* + */
#define MINUS      26 /* - */
#define SCOLON     27 /* ; */
#define GRTER      28 /* > */
#define NOT        32 /* ! */
#define MULTIP     34 /* * */
#define COMMENT    35 /* // */
#define LBRACKET   36 /* { */
#define RBRACKET   37 /* } */

```

It's also necessary to define the token type and a "to string" function (in this case, tok2str) in **token.h**.

```

/*----- Token type definition and toString function -----*/
typedef struct {          /* One lexical token: */
    int kind;            /* Lexical class */
    char id[MAX_ID_LENGTH]; /* if kind is TOK_ID, the actual identifier */
    int val;             /* if kind is TOK_INTGR, the int constant value */
} Token;

/* Store a string representation of Token t in s. */
/* pre: s must have enough space for the result. */
void tok2str(Token t, char* s);

```

The implementation of the token functions should be very straightforward. The key is to

implement the tok2str (Token t, char* s) function for different tokens. The following code fragment shows examples of cases of different tokens.

```
switch (t.kind) {
    case TOK_EOF: strcpy(s, "EOF"); return; //EOF
    case TOK_INTGR: //integer constants
        sprintf(s, "%s%d%s", "INT(", t.val, ")"); return;
    case TOK_IF:  strcpy(s, "IF");  return; //“if” keyword
    ...
}
```

When the implementation of the tokens is complete, the work of implementing the *scanner* can begin. The following code fragment shows the header file for the *scanner*. The only goal of the *scanner* in this project is to process the source code in the form of a character stream as input, ignore any comments and white spaces, and then produce a token stream as output. As a result, the key function of the implementation of the *scanner* will be the “next_token()” function, as shown in the header file.

```
#include "token.h"
#ifndef SCAN_H
#define SCAN_H
/* Return the next token from the source program, or EOF if */
/* there are no more tokens in the input */
Token next_token();
#endif
```

The next_token() function should be straightforward. The basic steps are as follows:

1. Declare a local variable of type Token to store the returning token.
2. See what the next character is (ignore comments and white spaces), and then return the token with the appropriate “kind”, “id”, or “val” values:
 - a. If the next character is an operator, punctuation, or delimiter, return the appropriate kind (e.g., if the character is “(”, return the token with the “kind” value of LPAREN);
 - b. If it is a number (integer), return the token with the “kind” value of TOK_INTGR and the “val” value of the whole number, but not just the first digit; and
 - c. If it is a word (string/character array), return the token with the “kind” value of TOK_ID and the “id” value of the whole string, but not just the first character. If the string is a keyword (e.g., “if”, “else”, etc.), return the token with the appropriate “kind” value. For example, if the keyword is “if”, the “kind” value for the returning token should be TOK_IF.

The following code fragment shows part of the scanner source code as an example for processing strings (identifiers).

```

Token next_token()
{ char nextchar; Token result;
  int count = 0;
  char* wholestring; //storing the string which is to be processed
  skip_space_comment();//for skipping any comment or white space
  nextchar = next_char();//get the next character from source
  result.kind = TOK_EOF; //initialize result.kind to EOF
  wholestring = (char*) malloc(MAX_ID_INT_LENGTH);//allocate memory
space for wholestring.
  switch(nextchar){
    /*Code for different cases for the next character. Only providing
one example for processing strings.*/
case 'a': case 'b': case 'c': ... .. case 'x': case 'y': case 'z':
case 'A': case 'B': case 'C': ... .. case 'X': case 'Y': case 'Z':
    //if the next characters are letters, digit or _, add to wholestring
    while(isalpha(nextchar) || isdigit(nextchar) || nextchar == '_'){
      wholestring[count] = nextchar;
      count++;
      current++;
      nextchar = next_char();
    }
    wholestring[count] = '\0';
    //these conditions are to distinct keywords from identifiers.
    //assume there is an array named keywords storing the keywords.
    if(find(wholestring, keywords) != 0){
      if(strcmp(wholestring, "if") == 0){result.kind = TOK_IF;}
      if(strcmp(wholestring, "else")==0){result.kind = TOK_ELSE;}
      if(strcmp(wholestring, "int")==0){result.kind = TOK_INT;}
      if(strcmp(wholestring, "while")==0){result.kind = TOK_WHILE;}
      if(strcmp(wholestring, "return")==0){result.kind =
TOK_RETURN;}
    }else{
      result.kind = TOK_ID;
      strcpy(result.id, wholestring);
    }
    break;
  }
  free(wholestring);
  return result;
}

```

The method of processing comments and white spaces is not that straightforward. There may be many different ways to do it. In this project, a recursive way was chosen. As seen in the previous code fragment, a function called `skip_space_comment()` should be implemented when

processing comments and white spaces.

The overall idea of this function is when the current character is a comment or white space:

1. If the current character is not at the end of the current line, just skip the current character and read the next character.
2. Use separate cases for white spaces and comments.
3. For white spaces, if the next character is also a white space, call `skip_space_comment()` recursively.
4. For comments, if the next character is also a `"/"` character, skip the whole line, and call `skip_space_comment()` recursively.
5. Process the end of the line appropriately.

Parser and Code Generation:

Every programming language has grammar rules. Parsing a sentence requires actually finding a derivation of the sentence according to grammar rules. In the parsing part, a syntax tree (also called a parsing tree) should be built, and then a target code should be generated. For actual compilers in the industry, there are many advantages to building an explicit parsing tree. But in this project, a parsing tree will be built implicitly in a recursive manner.

The following code sample shows how our parser can be implemented.

```
//start symbol.
void program()
{
    ...
    current_token = next_token();
    while(current_token.kind != TOK_EOF)
    {
        func();
    }
    ...
}

//parsing while statement
void while_stmt(symtab* st_local)
{
    ...
    current_token = next_token(); //skip while.
    current_token = next_token(); //skip "("
    bool_expr(st_local, lbl_after_while); //parse the boolean expression
    current_token = next_token(); //skip ")", so current_token here is the stmt.
    stmt(st_local); //parse the statement
    ...
}
```

What the parser actually does – except building the parsing tree – might be confusing, but the answer is that it does nothing more than that. At the time it is building the parsing tree, it is going to each level of the program, and generating the appropriate target code at each level's code generation part.

There is nothing complex in the code generation part. It actually simply uses the “printf()” function to print the correct target codes in the correct places.

The next code sample is an example of parsing while providing a statement with the code generation for the X86 assembly code.

```
//parsing while statement, with code generation.
void while_stmt(symtab* st_local)
{
    char * lbl_before_while = malloc(MAX_ID_INT_LENGTH);
    char * lbl_after_while = malloc(MAX_ID_INT_LENGTH);
    current_token = next_token();//skip while.
    current_token = next_token();//skip "("

    //To create a label (e.g. L1, for the assembly language),
    //glb_lbl is a variable declared in the very beginning of the parser.
    new_label(glb_lbl);
    strcpy(lbl_before_while, glb_lbl);
    printf("%s %s\n", lbl_before_while, ":");
    new_label(glb_lbl);
    strcpy(lbl_after_while, glb_lbl);
    bool_expr(st_local, lbl_after_while); //parse the boolean expression
    current_token = next_token();//skip ")", so current_token here is the stmt.
    stmt(st_local); //parse the statement
    printf("\tjmp\t"); //jmp keyword in the assembly language
    gen(lbl_before_while);
    gen(strcat(lbl_after_while, ":"));
    exit_program("while_stmt");
    free(lbl_before_while);
    free(lbl_after_while);
}
```

Conclusions:

Developing a compiler for a programming language requires a lot of work. This document provides a general idea of the structure of basic compilers and the very basic process of building a compiler. Just to review, a compiler consists of a scanner and a parser, and generates a target

code from a source code. The scanner processes the source code, eliminates white spaces and comments, and then generates tokens for the parser to use. The parser builds the parsing tree. All the parsing processes of the compiler will need to follow a set of rules, which is usually specified by the programming language grammar. The actual job of building a compiler will be more difficult than it appears in this document. Many problems will be occur and need to be solved in the developing process. For example, in the parser part, an ambiguous problem needs to be solved in order to parse different programs with similar source codes correctly; for example, $3-4+5$ and $3-(4+5)$ should have different results when parsed correctly. The code samples provided in this document come from an actual project, but have been edited to fit this document. They are provided for informational purposes only, without any warranty, and may not be compiled successfully.