CS215 Compiler Project Report

Yudong Luo (SID: 5140309166)
Department of Computer Science and Technology
Shanghai Jiao Tong University
Shanghai, China
Email: miyunluo@foxmail.com

***Abstract*—CS215 Complier Project requires students to implement a complier for translating Small-C source code into MIPS assembly code. In this project, I construct the abstract syntax tree and translate the syntex tree into intermediate code, finally I interpret the intermediate code into MIPS code. I note that I implementd the optimization part, which is a bonus point. After this project, I get a Small-C complier and the MIPS code can be run on SPIM or other simulator. This report will clearly describe how to implement this Small-C compiler.**

I. Introduction

This compiler consists of five parts, including lexical analysis, syntax analysis, intermediate-code generation, optimization and machine-code generation. The compiler construction is shown in Figure 1. Here comes the introduction of these five phases in the construction figure, and the design detials comes in the following sections.

1) **Lexical Analysis:**
   This compiler begins with lexical analysis. The lexical analyzer generates tokens. It reads the source codes of Small-C and split the characters into meaningful lexemes. For each lexeme, the lexical analyzer generate a token in the following formation

   $$< token - name, attribute - value >$$

   Flex (fast lexical analyzer generator) is used to generate the lexical analyzer. Flex is easy to install on ubuntu linux. The details of Flex and description of how I use Flex to generate the lexical analyzer comes in Section II;
2) **Syntax Analysis:**
   The second part of the compiler is syntax analysis or another name called parsing. The parser uses the first component in the token pair produced by the lexical analyzer to create a tree-like intermediate representation that gives the grammatical structure of the token stream. The general format of the representation is a syntax tree, whose interior node represents an operation and the children of this node represent the arguments of this operation. I generate a syntax analyzer to build a syntex tree from token stream.
   Yacc (yet another compiler-compiler),which is an LALR parser generator, is used here to generate the syntex tree. More details are covered in Section III;
3) **Intermediate-code Generation:**
   This part includes Semantic Analysis and Intermediate Representation.

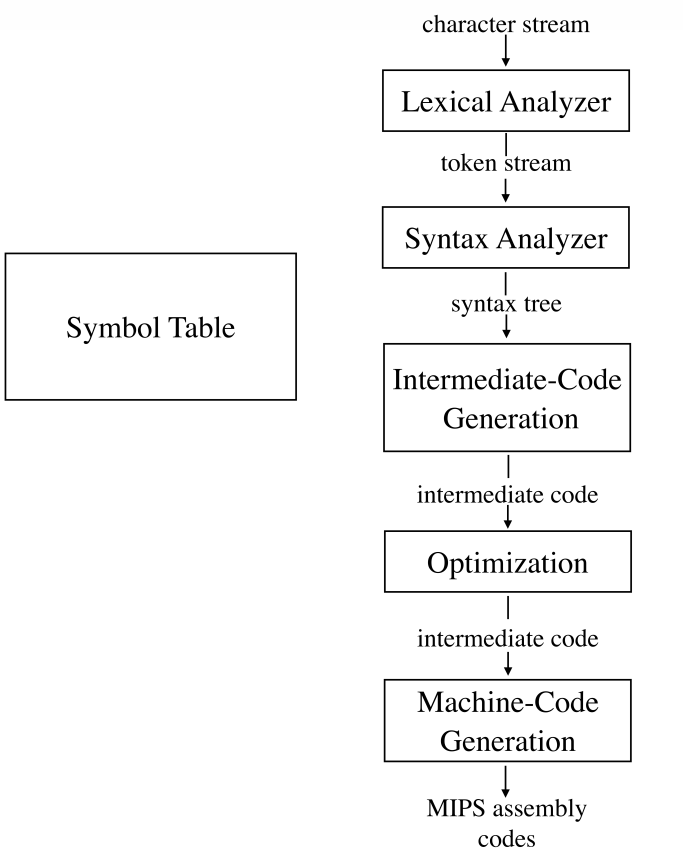

Fig. 1. Phases of a compiler

   The semantic analyzer uses the syntax tree generated by syntax analyzer and the symbols in the symbol table to check the sematic consistency of source program with the language definition. I implement some important semantic analysis features, and you can refer Section IV for details. In Section IV, I not only list which feature I have implemented but also clearly describe how I implement that.
   Intermediate Representation is a kind of language which is simple, clear and independent of the details of both source language and target language. The IR is write to a file for further optimization in the following part.
4) **Optimization**
   The code optimization part aims to optimize the inter-

mediate code so that better target code will be generated. I implement some important optimization features, and you can refer Section V for details. In section V, I not only list which feature I have implemented but also clearly describe how I implement that.

5) **Machine-Code Generation:**
The last task remained to be done is machine-code generation, or rather, translating the intermediate representation into target codes. In this project, **MIPS** assembly code is orderd to be the target language. Here, I split code generation into two parts. The first part is instruction selection, and the second part is register allocation.
Although most of the instruction selections are done in the Intermediate Representation part, there are still some thing can be done in the instruction selection part, such as checking labels and branches and so on. In this part I just assume all the variables are stored in registers, and the only thing I need to do is to choose suitable instructions and translate IR into **MIPS** assembly codes. In register allocation part, a register allocation algorith is implemented. More details about instruction selection and register allocation are given in Section VI;

In this report, I will first introduce the design of lexical analyzer in Section II. Then, I will present the syntax analyzer in Section III. And in the following sections, I will cover intermediate-code generation in Section IV, optimization in Section V, and machine-code generation in Section VI. Finally, I conclude this report in Section VII;

II. Lexical Analysis

Flex is used as a tool to implement the lexical analyzer, so I will concisely introduce Flex first. Then I will show you how to generate the lexical analyzer using Flex.

A. Introduction to Flex

Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc (refer to Section III) parser generator. Lex, originally written by Mike Lesk and Eric Schmidt[1] and described in 1975, [2][3] is the standard lexical analyzer generator on many Unix systems, and an equivalent tool is specified as part of the POSIX standard. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

B. Lexical Analyzer Generation

Figure 2 is a usage paradigm of Lex.

Lex source is a table of regular expressions and corresponding program fragments. The recognition of the expressions is performed by a deterministic finite automation. At each expression appears in the input, the corresponding fragment is executed.

A lex input file consists of three sections: definition, rules, and user subroutines, which are separated by %%. The first and second part must exist, but may be empty, the third part and the second %% are optional.

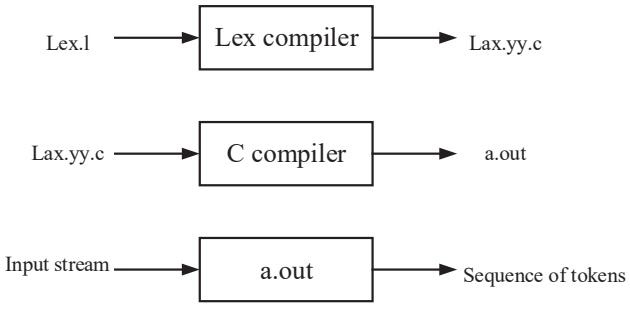

Fig. 2. Phases of a compiler

A part of rules in lex source is shown as follow:

"(" {yylval.string = strdup(yytext); return LP;}
")" {yylval.string = strdup(yytext); return RP;}
"[" {yylval.string = strdup(yytext); return LB;}
"]" {yylval.string = strdup(yytext); return RB;}
"{" {yylval.string = strdup(yytext); return LC;}
"}" {yylval.string = strdup(yytext); return RC;}

III. Syntax Analysis

Yacc is used as a tool to implement the syntax analyzer. In this section, I will concisely introduce Yacc first, then I will show you how to generate the syntax analyzer using Yacc.

A. Introduction to Yacc

YACC is an acronym of "Yet Another Compiler Compiler". It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF.[4] It was originally developed in the early 1970s by Stephen C. Johnson at AT&T Corporation and written in the B programming language, but soon rewritten in C.[5] It appeared as part of Version 3 Unix,[6] and a full description of Yacc was published in 1975.[7]

B. Syntax Analyzer Generation

Yacc is an LALR parser generator, which can generate tables according to the grammar rules, driver routines in C programming language and y.output a report file. Yacc can be considered as a compiler compiler. It need to cooperate with lex, which will invoke yylex() automatically and generate *y.tab.h* file. For each token that lex recognized, a number is returned (from yylex() function).

A Yacc input file consists of three sections: definition, rules, and user code, which are separated by %%. All terminal symbols are declared through %token in definition section.

C. *Handling Conflicts and Error*

We note that it is important to specify operator precedence and associativity to avoid conflicts. For example, *"IF LP EXP RP STMT"* should have lower precedence than *"IF LP EXP RP STMT ELSE STMT"*. The solution is shown as follows, some changes of the semantic rule will be explained in Section IV.

```
STMT:
  | IF LP EXPS RP STMT %prec IFX
  | IF LP EXPS RP STMT ELSE STMT %prec ELSE
```

As for syntax error detection, I write the corresponding error message in yyerror(). The error message which may be a missing symbol, unrecognized word or imcomplete expression, will be printed out with the corresponding line number. The format of error detection in yyerror() is shown as follows:

```
void yyerror(char* s)
{
  fflush(stdout);
  fprintf(stderr,...);
}
```

IV. Intermediate-Code Generation

This part consists of semantic analysis and intermediate representation.

The semantic analyzer uses the syntax tree generated by syntax analyzer and the symbols in the symbol table to check the sematic consistency of source program with the language definition. So I will show the implementation of symbol table first.

A. Symbol Table

Symbol table is the data structure used by compilers to hold information about source program. For symbols in the symbol table, I construct struct `SymbolTable` to record the related information. Each ID has a symbol in symbol table. When we want to declare a new ID, we create a symbol. When we want to use an ID, we need to check whether it is already in the symbol table. A concise defination of struct `SymbolTable` in *Semantic.h* is as follows:

```
struct SymbolTable{
  map<...> table;
  map<...> struct_table;
  map<...> struct_id_table;
  map<...> struct_name_width_table;
  map<...> width_table;
  map<...> array_size_table;
  int parent_index;
};
```

Here comes the meaning of the data members in class `SymbolTable`:

- *table:* maps variable names to types, eithor int or struct.
- *struct_table:* maps variables of a struct to a vector which contains the name of the members of the struct varaible.
- *struct_id_table:* maps the name of the type of the struct to a vector which contains the name of the members of this type of struct.
- *struct_name_width_table:* maps name of struct variables to the number of members of this struct, which is later used in the code generation part.
- *array_size_table:* maps name of arrays to a vector which contains how many 4 bytes of the width each dimension.
- *parent_index:* records the location of the upper namespace.

I note that map in STL is used in this struct as an efficient hash table. Using STL in C plus plus is convenient and totally competent for the test cases. It is also easy to extent.

B. Semantic Analysis

After introducing symbol tables, we are ready to do the semantic checking. To make semantic analysis easier, some sematic rules are modified to prevent the case EXP $\rightarrow \epsilon$. That is I change EXP to EXPS, which is shown in Section II. In general, after getting the syntax tree from the syntax analyzer, we should traverse the tree from root to generate the symbol table and do the semantic checking. I will show the specific methods I used to check the semantic errors and the implementation idea:

- *Variables and functions should be declared before usage.*
  When we want to use a variable, we need to check whether it is already in the symbol table. As we can observe, the inner variable must be declared later. So we only need to check in the inverted order, and the first valid symbol we find is what we want. So when we get the entry, we can implement a function to search its current namespace to find if names match, otherwise it will go to the parent namespace, until the global scope.
  A notion is that when we leave a statement block, all the variables declared in this scope should be set to invalid.
- *Variables and functions should not be re-declared.*
  When we want to declare a new variable, we use its first letter to get the right entry, and check whether there is a valid symbol of the same scope having the same name. If not, we we can create a new symbol, otherwise an error accurs.
- *Reserved words can not be used as identifiers.*
  The function named isReserved() to check whether name of id coincides with reserved words.
- *Program must contain a function int main() to be the entrance.*
  The main() function is the general entrance for C-like language. A new variable named bool_main is the flag of the existence of main(). It is initialized to be false. After the traversal of the syntax tree, we will check bool_main, if its false, we will report error.
- *The number and type of variable(s) passed should match the definition of the function.*

When we want to check parameters of a function, we need to record the number of arguments of a function call and check whether they match the defination. I note that if a parameter of a function is also a function call, we should store parameters temporarily and handle them afterwards. You can refer to *func_cnt_table* for more detials.

- *Use [] operator to a non-array variable is not allowed.* The member of struct named width_table is used to handle this problem. When we meet"[]" operator, we check whether the variable is a struct first, and how many 4 bytes it has second. If the number is one, we consider it is not an array.
  However, some reasonable defination like *a[1]* fails with this method.
- *The . operator can only be used to a struct variable.* When we meet dot operator, we check whether the variable is a struct first, and get its declaration namespace second. Then we refer to *struct_id_table* and check whether the id after dot is a member of that struct.
- *Break and continue can only be used in a for-loop.* I add a new variable named *in_for* as the flag of wherther we are in the loop. When we enter a for loop, the value of *in_for* is modified to be true and when we exit the loop, it is restored to be false.
- *Right-value can not be assigned by any value or expression.* Because the limitation of Small-C, the number of expressions that could be a left value is limited, such as ID ARRS, EXPS ASSIGN EXPS and etc... When we are doing assigning actions or calling read() function, we can check whether it is within these kinds.
- *The condition of if statement should be an expression with int type.* We can check whether the EXPS is a struct first, and get the bytes of the id of the EXPS second, then we judge whether its an array pointer or an int.
- *The condition of for should be an expression with int type or ϵ.* This is the same as the former one.
- *Only expression with type int can be involved in arithmetic.* This is the same as the former one.
- *Other* Apart from the check requirements given in the project guidance, I also check whether the size of an arry is postive, and I ensure the members of a struct and the parameters of a function don't have the same name.

C. Intermediate Representation

After semantic analysis, we are ready to translate the syntax tree into IR. IR (Intermediate Representation) is a kind of language which is simple, clear, and independent of the details of both source language and target language, so that it acts as a bridge between high-level language and machine language and can be easily optimized. I choose the three-address code as the format of IR in this project.

My IR is quite analogous to machine code. I note that I assume registers are sufficient. The benefit is that I can assign a register to a variable declared.
Part of my IR and its three-address code format is as follows, you can refer to *Intermediate.h* for more details and I will give the full IR design in a separate file.

...
or,a,b,c $a=b|c$
xor,a,b,c $a=b\hat{}$
and,a,b,c $a=b\&\ c$
sll,a,b,c $a=b<<c$
srl,a,b,c $a=b>>c$
add,a,b,c $a=b+c$
sub,a,b,c $a=b-c$
mul,a,b,c $a=b*c$
div,a,b,c $a=b/c$
rem,a,b,c $a=b\%c$
neg,a,b $a=-b$
...

V. Optimization(Optional)

The code optimization phase aims to improve the intermediate representation so that better target code will be generated.

A. Dead code elimination

We can eliminate dead code in this method. Instructions that compute a value never used can be deleted. The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

For example, instruction *mov,t2,t1* and *mov,t3,t2* can be replaced by *mov,t3,t1*. You can refer to *Optim.h* for more detials.

B. Algebraic reduction

We can apply arithmetic identities, such as

$$a + 0 = 0 + a = a \quad a - 0 = a$$

to eliminate computations from a basic block.

There exist some other optimization methods. However, I cannot implement all of them due to the limitation of time. I plan to finish them in the future work.

VI. Machine-code generation

The last thing that need to do is to translate the intermediate representation into target machine codes. In this project, **MIPS** assembly code is the target language. This section consists of two parts, instruction selection and register allocation.

A. Instruction Selection

Because my IR is analogous to machine code, most instruction selection is implemented in Section IV, and the rest task is implemented in this part.

- Code abbreviation
  Delete branch if destination is immediately followed.
  Replace the register with $0 wherever the value zero is used.
  For all arithmetic expressions that involves constants, interpret them to intermediate instructions
  Use sll when times 4.
- Register vs. Immediate
  Exp() can return a register or an immediate number, which need to be handled carefully. If its an immediate number, we need to first allocate an address to store it, then load it to a register. If its a register, we can just use it.

B. Register Allocation

Register allocation is to assign regesters to variables. In fact we can always use lw and sw operations to make things work even with just 3 registers. In this part, I use linear scan algorithm.

I calculate the use and def of each quadruple, and iteratively update the in and out of each quadruple according to the BFS flow gram, and finally allocate registers for each variable using the linear scan algorithm.

C. Code Generation

Since the IR is close to machine code and the register allocation part has been done, translation from IR to MIPS is quite straightforward. You can refer to *Code_gen.h* for more detials.

VII. Conclusion

This project is quite challenging. It is the longest program I have written, and it is the most difficult one. I am sure I have gained a lot deeper understanding for compiler and that my coding ability and engineering skills surely improved.

Thanks our teacher Dr. Jiang for giving us such a project to learn more about compiler principles. Thanks our T.A. for providing guide. Also thanks for my colleagues for giving me great help!

References

[1] Lesk, M.E.; Schmidt, E. *Lex – A Lexical Analyzer Generator* Retrieved August 16, 2010.
[2] Lesk, M.E.; Schmidt, E. *Lex – A Lexical Analyzer Generator* UNIX TIME-SHARING SYSTEM:UNIX PROGRAMMERS MANUAL, Seventh Edition, Volume 2B. bell-labs.com. Retrieved Dec 20, 2011.
[3] Lesk, M.E. *Lex – A Lexical Analyzer Generator* Comp. Sci. Tech. Rep. No. 39 (Murray Hill, New Jersey: Bell Laboratories).
[4] *The A-Z of Programming Languages: YACC* Computerworld. Retrieved 30 November 2012.
[5] Ritchie, Dennis M. *The Development of the C Language* Association for Computing Machinery, Inc.
[6] McIlroy, M. D. *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971C1986* CSTR. Bell Labs. 139.
[7] Johnson, Stephen C. *Yacc: Yet Another Compiler-Compiler* AT&T Bell Laboratories Technical Reports (AT&T Bell Laboratories Murray Hill, New Jersey 07974) (32). Retrieved 31 October 2014.