SMALLC Complier

Project 2 Report

胡泾莓 5120309147

Outline

Project Name

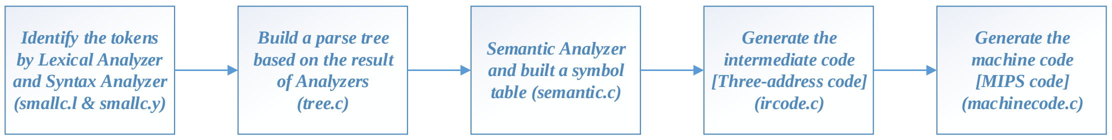
SMALLC Complier

Project Objective

- Design and implement a simplified compiler, for a given programming language, namely SMALLC, which is a simplified C-liked language containing only the core part of C language.
- Achieve a lexical analyzer, a parser for parse tree generation, a parse tree to check the semantic errors, and then translate the parse tree into an intermediate representation (IR). Finally, generate the target assembly codes.

Description

Design Architecture


Identify the tokens by Lexical Analyzer and Syntax Analyzer (smallc.l & smallc.y)
Build a parse tree based on the result of Analyzers (tree.c)
Semantic Analyzer and built a symbol table (semantic.c)
Generate the intermediate code [Three-address code] (ircode.c)
Generate the machine code [MIPS code] (machinecode.c)

Lexical Analyzer

%{

C declarations and includes

%}

```
Declarations
%%
Translation rules
%%
User subroutines
```

Syntax Analyzer

```
%{
    C declarations and includes
%}
Token definitions
%%
Syntax rules
%%
User codes
```

Lexical Analyzer and *Syntax Analyzer*, both of them have already finished in Project 1. In order to make the grammars more clear and concise, we choose to improve the grammar. For example, we add DEC into the grammar, as followed:

```
DEC   ->  VAR
       |  VAR ASSIGN INIT
```

Parse Tree

When *smallc.y* judges the tokens in the text files, it adds the nonterminal and terminal nodes into the parse tree and establish the relationships of each node. We let the parse tree to be a son-brother tree in order to make sure the following steps will work well.

In *tree.c*, we separate apart terminal and nonterminal symbols and use child and sibling to distinguish the type of reduce grammar. The first token of the reduce grammar is always the child of the parent node in parse tree, and the other tokens are regarded as the sibling of this child.

In this way, we generate a parse tree and print it into an output file.

Semantic Analysis

After the generation of parse tree, we built the symbol table using data structure named hash table. Each node includes the name of function/ structure/ array/ variable, the type/ width/ address of function /structure/ array/ variable and some information else.

Create the corresponding function for each kind of type such as variables, arrays, structures and functions, and then use them in semantic analysis.

As for *Semantic Analysis*, recursion is the most significant part. We do semantic analysis from the root node of parse tree. For each reduce type we defined in *Syntax Analyzer*, do semantic analysis one by one. And check the error information at the same time. We define twenty kinds of error messages in our compiler, as followed:

"Undefined Variable"
"Undefined Function"
"Undefined Struct"
"Multidefined Variable"
"Multidefined Function"
"Multidefined Struct"
"Incompatible Types when 'assign'"
"Incompatible Types"
"Incompatible Type when 'return'"
"Incompatible Function Parameter"
"Incompatible Function Call"
"Incompatible Struct Type"
"Left Value Required"
"Array Type Required"
"Invalid Array Index"

"Invalid Struct Scope"
"Invalid Struct Definition"
"Function Declared No Defined"
"Mismatched Function"
"Invalid Condition"

The usage of *Semantic Analysis* in my compiler is to detect the semantic error of test files. Almost the same work will be done in intermediate code generation part. But in order to make our compiler work clear, we separate those works into several independent parts.

Intermediate Representation

In this part, we do not need to check the error in test files, the intermediate code parse is very similarity as semantic analysis parse. But we add some other operations into parse.

In *Intermediate Representation*, we create a link list with its header named *irheader*. When we meet a parse tree node, we do intermediate code parse and insert the corresponding intermediate code into the tail of the link list.

The most difficult part in this section is that the handling methods of those variables and arrays which are declared without initialization. We use a integer variable in each node structure to store whether it is be initialized or not. The following steps will be achieved in *Machine Code Generation*.

As for output, we have already created a link list for intermediate code. We can judge the node type of each node in link list, then do its corresponding instruction.

Machine Code Generation

We first define the machine code we will use in this part, as followed:

```
#define INSTRUCTION_LW      "\tlw %s, %d(%s)\n"
#define INSTRUCTION_SW      "\tsw %s, %d(%s)\n"
#define INSTRUCTION_LI      "\tli %s, %d\n"
#define INSTRUCTION_ADD     "\tadd %s, %s, %s\n"
#define INSTRUCTION_MUL     "\tmul %s, %s, %s\n"
#define INSTRUCTION_SUB     "\tsub %s, %s, %s\n"
#define INSTRUCTION_DIV     "\tdiv %s, %s, %s\n"
#define INSTRUCTION_MOD     "\trem %s, %s, %s\n"
#define INSTRUCTION_AND     "\tand %s, %s, %s\n"
#define INSTRUCTION_OR      "\tor %s, %s, %s\n"
#define INSTRUCTION_XOR     "\txor %s, %s, %s\n"
#define INSTRUCTION_NOT     "\tnor %s, %s, %s\n"
#define INSTRUCTION_SLL     "\tsll %s, %s, %d\n"
#define INSTRUCTION_SRL     "\tsrl %s, %s, %d\n"
#define INSTRUCTION_JAL     "\tjal %s\n"
#define INSTRUCTION_JR      "\tjr %s\n"
#define INSTRUCTION_J       "\tj "LABEL_PREFIX"%d\n"
#define INSTRUCTION_FLBL    "%s:\n"
#define INSTRUCTION_MOV     "\tmove %s, %s\n"
#define INSTRUCTION_RET     "\tjr %s\n"
#define INSTRUCTION_BEQ     "\tbeq %s, %s, "LABEL_PREFIX"%d\n"
#define INSTRUCTION_BNE     "\tbne %s, %s, "LABEL_PREFIX"%d\n"
#define INSTRUCTION_BGT     "\tbgt %s, %s, "LABEL_PREFIX"%d\n"
#define INSTRUCTION_BLT     "\tblt %s, %s, "LABEL_PREFIX"%d\n"
#define INSTRUCTION_BGE     "\tbge %s, %s, "LABEL_PREFIX"%d\n"
#define INSTRUCTION_BLE     "\tble %s, %s, "LABEL_PREFIX"%d\n"
```

And also we define *ASSEMBLY_HEADER* as a constant in order to print out immediately, as followed:

```
#define ASSEMBLY_HEADER \
".data\n" \
"_prompt: .asciiz \"Enter an integer: \"\n" \
"_ret: .asciiz \"\\n\" \n" \
".globl main \n" \
".text \n" \
"read: \n" \
"\tli $v0, 4 \n"\
"\tla $a0, _prompt \n"\
"\tsyscall \n"\
"\tli $v0, 5 \n"\
"\tsyscall \n"\
"\tjr $ra \n"\
"write: \n"\
"\tli $v0, 1 \n"\
"\tsyscall \n"\
"\tli $v0, 4 \n"\
"\tla $a0, _ret \n"\
"\tsyscall\n"\
"\tmove $v0, $0\n"\
"\tjr $ra\n"
```

For each kind of assembly instruction, we create a function to combine the defined instruction which miss exactly register number or immediate operand and those certain messages together.

In *Machine Code Generation*, we should firstly set enough place for the intermediate variables to store in the stack.

The most difficult part in this section is the handling method of the global variables. We create another link list to store the global variables. And if a variable we find in function is not defined in this function scope, we will search it in the global variable link list. If this variable is a global one, the instructions which contain it will be changed.

In this compiler, the global variables are also stored in stack, when we use these, we will get their offsets which are calculated when the stack pointer is on the primitive place. But as the program processing, the stack pointer changes when we call a function or in some other cases. The offsets of those global variable will be totally wrong because the stack pointer *$sp* is not at its primitive place at all.

In that way, we firstly store the value of *$sp* into an empty register. And then, when we use the global variable, we add some other instruction to let the stack pointer be its primitive place momentarily. After we finished the usage of those, we set the stack pointer back to its correct place. In this way, we can utilize the global variable without program reporting errors.

Then we do the corresponding operations for each node we create in *Intermediate Representation*.

Result

After the several weeks hard working, I finally make my code work through 3 basic test cases. And in my own test, my code cannot be passed through a very few cases. The running results and other details about the code design will be delivered in the demo.

Acknowledge

This compiler project may be the largest one I have ever done in the last two years. I have attained a much more clear view about how the compiler works. Many thanks to TA and my fellow students who gave me hundreds of suggestion and countermeasure.