Project 2: SMALLC Complier

CS215
Shanghai Jiao Tong University

November 23, 2014

1 Introduction

C and some *C-liked* languages are the dominant programming languages. In this project, you are required to 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. After finishing this project, your coding ability will surely improve. Also, you can get a compiler, which can translate SMALLC source codes to **MIPS** assembly codes. These assembly codes can run on the **SPIM** simulator, or you can assemble it to machine code to run it on a real computer. In this project, Linux environment (e.g., Ubuntu or CentOS) is required.

To build a whole compiler may seem to be a huge, difficult, and complicated task (and actually it is). To make this project easier, we divide the procedure into 5 parts. The first part, in Section 2, is to write a lexical analyser to split the source codes into tokens. The second part, in Section 3, is to write a parser for parse tree generation. The third part, in Section 4, you are required to check the parse tree to make sure that there are no semantic errors. Then we translate the parse tree into an intermediate representation (IR). The forth part, in Section 5, which is optional, requires you to optimize the IR. And the last part, in Section 6, is to generate the target assembly codes.

2 Lexical Analyzer

In this section, you are going to write a lexical analyser. The lexical analyser reads the source codes of SMALLC and separates them into tokens.

2.1 Tokens

INT	⇒	/* integer [1]*/
ID	⇒	/* identifier[2]*/
SEMI	⇒	;
COMMA	⇒	,
BINARYOP	⇒	/* binary operators[3] */
UNARYOP	⇒	/* unary operators[4] */
TYPE	⇒	int
LP	⇒	(
RP	⇒	)
LB	⇒	[
RB	⇒	]
LC	⇒	{
RC	⇒	}
STRUCT	⇒	struct
RETURN	⇒	return
IF	⇒	if
ELSE	⇒	else
BREAK	⇒	break
CONT	⇒	continue
FOR	⇒	for

2.2 Operators

Operators in SMALLC are shown below.

Precedence	Operator	Associativity	Description
1	()	Left-to-right	Function call or parenthesis
	[]		Array subscripting
	.		Structure element selection by reference
2	−	Right-to-left	Unary minus
	!		Logical NOT
	++		Prefix increment
	−−		Prefix decrement

[1] A sequence of digits or digits followed by "0x(0X)" or "0" without spaces. In addition, the value should be in the range of $(-2^{31}, 2^{31})$

[2] A character string consisting of alphabetic characters, digits and the underscore. In addition, digits can't be the first character.

[3] See section 2.2.

[4] See section 2.2.

	~		Bit NOT
3	*	Left-to-right	Product
	/		Division
	%		Modulus
4	+		Plus
	−		Binary minus
5	<<		Shift left
	>>		Shift right
6	>		Greater than
	>=		Not less than
	<		Less than
	<=		Not greater than
7	==		Equal to
	! =		Not equal to
8	&		Bit AND
9	^		Bit XOR
10	|		Bit OR
11	&&		Logical AND
12	||		Logical OR
13	=	Right-to-left	Assign
	+=		+ and assign
	-=		- and assign
	*=		* and assign
	/=		/ and assign
	&=		& and assign
	^=		^ and assign
	|=		| and assign
	<<=		<< and assign
	>>=		>> and assign

2.3 Flex

Flex is short for *fast lexical analyzer generator*, which is a free version of lex written in C. In this project, you can use flex to generate the lexical analyzer.

Here are some references about Flex:

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Second Edition. Chapter 3.5.
- http://en.wikipedia.org/wiki/Flex_lexical_analyser.

- http://flex.sourceforge.net/manual/.

3 Syntax Analyzer

In this step, you are going to perform syntax analysis using Yacc.

3.1 Grammar[5]

PROGRAM	→	EXTDEFS
EXTDEFS	→	EXTDEF EXTDEFS
	|	ϵ
EXTDEF	→	TYPE EXTVARS SEMI
	|	STSPEC SEXTVARS SEMI
	|	TYPE FUNC STMTBLOCK
SEXTVARS	→	ID
	|	ID COMMA SEXTVARS
	|	ϵ
EXTVARS	→	VAR
	|	VAR ASSIGN INIT
	|	VAR COMMA EXTVARS
	|	VAR ASSIGN INIT COMMA EXTVARS
	|	ϵ
STSPEC	→	STRUCT ID LC SDEFS RC
	|	STRUCT LC SDEFS RC
	|	STRUCT ID
FUNC	→	ID LP PARAS RP
PARAS	→	TYPE ID COMMA PARAS
	|	TYPE ID
	|	ϵ
STMTBLOCK	→	LC DEFS STMTS RC
STMTS	→	STMT STMTS
	|	ϵ
STMT	→	EXP SEMI
	|	STMTBLOCK
	|	RETURN EXP SEMI
	|	IF LP EXP RP STMT
	|	IF LP EXP RP STMT ELSE STMT
	|	FOR LP EXP SEMI EXP SEMI EXP RP STMT

[5]Input and output are not included, which will be provided in 6.3

	|	CONT SEMI
	|	BREAK SEMI
DEFS	→	TYPE DECS SEMI DEFS
	|	STSPEC SDECS SEMI DEFS
	|	ϵ
SDEFS	→	TYPE SDECS SEMI SDEFS
	|	ϵ
SDECS	→	ID COMMA SDECS
	|	ID
DECS	→	VAR
	|	VAR COMMA DECS
	|	VAR ASSIGN INIT COMMA DECS
	|	VAR ASSIGN INIT
VAR	→	ID
	|	VAR LB INT RB
INIT	→	EXP
	|	LC ARGS RC
EXP	→	EXPS
	|	ϵ
EXPS	→	EXPS BINARYOP EXPS
	|	UNARYOP EXPS
	|	LP EXPS RP
	|	ID LP ARGS RP
	|	ID ARRS
	|	ID DOT ID
	|	INT
ARRS	→	LB EXP RB ARRS
	|	ϵ
ARGS	→	EXP COMMA ARGS
	|	EXP

3.2 Notes and Hints

- The meanings of statements in SMALLC is based on the meanings in *C*.
- The grammar given above may induce one or two reduce-reduce/shift-reduce conflicts, you need to assign precedence of some expressions manually to eliminate these conflicts. For example, "IF LP EXP RP STMT" should have lower precedence than "IF LP EXP RP STMT ELSE STMT".
- A number starts with '0x' or '0X' is a hexadecimal number, while a

number starts with '0' is a octal number.

- Only integers and 1-dimensional array can be initialized.
- The dimension of arrays will be no more than 2.
- *Struct* can only contain *int* variables.
- The return type of a function can only be *int*.
- There are no "strange" statements such as $a - - - b$.

3.3 Yacc

Yacc is an LALR parser generator, which stands for *yet another compiler-compiler*. In this project, we can use yacc/bison (bison is another version of yacc) to generate a parser.

Here are some references about Yacc:

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Second Edition. Chapter 4.9.
- http://en.wikipedia.org/wiki/Yacc.
- http://www.gnu.org/software/bison/manual/.

4 Intermediate-code Generation

4.1 Semantic Analysis

Since not every program that matches the grammar is the correct program, you need to do semantic analysis and syntactic checking to examine potential semantic errors after generating a parse tree. For example, if a is an integer, and there is a statement "$a.b = 1$" in the program, your compiler need to report an error.

Different from the previous parts, you don't have any tool to rely on, all the codes should be written by yourselves. There are a number of things we need to do.

Specifically, we need to check the following things.

- Variables and functions should be declared before usage.
- Variables and functions should not be re-declared.

- Reserved words can not be used as identifiers. Reserved words can be found in TOKENs.
- Program must contain a function *int main*() to be the entrance.
- The number and type of variable(s) passed should match the definition of the function.
- Use [] operator to a non-array variable is not allowed.
- The . operator can only be used to a struct variable.
- *break* and *continue* can only be used in a for-loop.
- Right-value can not be assigned by any value or expression.
- The condition of *if* statement should be an expression with *int* type.
- The condition of *for* should be an expression with *int* type or ϵ.
- Only expression with type *int* can be involved in arithmetic.
- etc.

You are expected to implement as many of them as possible, and describe your implementations in the report.

4.2 Intermediate Representation

After semantic checking (or at the same time), you need to translate the parse tree into an intermediate representation (IR) and store your IR language into a file. An IR 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. In this project, you can choose parse tree or three-address code (recommended) to be the IR. Here is an example.

	i = 0
j = 0;	j = 0
for (i = 0; i < 5; i ++)	branch i ≥ 5 L1
j += i;	j = j + i
	L1:

Table 4: An example.

In the example, we translate the source codes into three-address codes. Then we can translate the IR into lower-level language without source language.

5 Optimization (Optional)

This section, which is optional and will be counted as a bonus, requires you to perform IR optimization (please refer to our textbook). There are different ways to optimize, for example:

- Common subexpression elimination
- Dead code elimination
- etc.

6 Machine-code Generation

The last thing you need to do is to translate the IR into target language. In our project, we choose the **MIPS** assembly language to be the target language. Then we will use **MIPS** simulator named **SPIM** [6] to simulate the assembly code.

We split code generation into two parts. The first part is instruction selection, and the second part is register allocation.

6.1 Instruction Selection

In this part, we assume that there are infinite registers. In other words, we don't need to consider which variables should be stored in registers and which variables should be stored in memory. Therefore, the only thing we need to do is to choose suitable instructions and translate the IR into **MIPS** assembly language. Note that there may be more than one ways to translate IR. For example, we can translate $a = b + 1$ into a *li* instruction and an *add* instruction, or just an *addi* instruction. Try to use an effective way when selecting the instruction.

[6] James R. Larus, *SPIM S20: A MIPS R2000 Simulator*

6.2 Register Allocation

You must have learnt some register allocation algorithms in this course. If needed, please refer to slides[7] for more details. After this step, we can obtain a whole compiler for SMALLC .

6.3 Input and Output

As we know, Input and output are important parts of a program. To test the compiler, we add two special functions to SMALLC :

- read(int x); input an integer to x,
- write(int x); output the value of integer x in one line.

You may need to change our token list and grammar to support these two functions, and then refer to the **SPIM** documentation to find out how **SPIM** hands I/O.

Now, you have finished your SMALLC compiler. Congratulations!

[7] X. Jia, *Register Allocation*

Submission Requirements

1. Pack the source files into a file named **StudentID-prj2.tar**. Please DO use "tar" command to compress your source files. Your source files should include but not limited to:

File name	Description
def.h	header file
smallc.l	Lex program
smallc.y	Yacc program
···	other code files
makefile	makefile
README	list the files in your submission
StudentID-report.pdf	project report

Table 5: File List.

2. Your analyzer will be tested by the following command:

```
./make
./scc "Source file name"
```

   Your analyzer (named "scc") is expected to read source SMALLC codes from a source file, and output the generated intermediate codes and MIPS codes into "InterCode" and "MIPSCode.s" respectively. Please output other information to *stderr*. Your generated MIPS codes will be tested on *SPIM* simulator.

3. Please state clearly the purpose of each program at the head of the source codes, and comment your programs if necessary.

4. A well-organized report in **PDF** format is really important. Since this project is really large, you need to illustrate clearly how you design and implement your compiler in your report.

5. Send your **StudentID-prj2.tar** file to cs215.sjtu@gmail.com.

6. Due date: Midnight, Dec. 19, 2014.

7. Project demo (optional): Dec. 20-21,2014.

8. Presentation (optional): TBA