Project 1: Implementing a Lexical-Syntax Analyzer

October 8, 2015

1 Introduction

In this project, you are expected to implement a simplified compiler frontend, including a lexical analyzer and a syntax analyzer, for Small-C , which is a C-like language containing a subset of the C programming language. You will learn how to incrementally design and implement the successive phases of the compilation process using off-the-shelf generators.

2 Step 1: Implementing a Lexical Analyzer

In this step, we will write a lexical analyser. The lexical analyser reads in the Small-C source code, and recognize tokens according to regular definitions.

2.1 Tokens

INT	$\Rightarrow$	/* integer [1]*/
ID	$\Rightarrow$	/* identifier[2]*/
SEMI	$\Rightarrow$	;
COMMA	$\Rightarrow$	,
DOT	$\Rightarrow$	.
BINARYOP	$\Rightarrow$	/* binary operators[3] */

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

[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.

UNARYOP $\Rightarrow$ /* unary operators[4] */
ASSIGNOP $\Rightarrow$ =
TYPE $\Rightarrow$ int
LP $\Rightarrow$ (
RP $\Rightarrow$)
LB $\Rightarrow$ [
RB $\Rightarrow$]
LC $\Rightarrow$ {
RC $\Rightarrow$ }
STRUCT $\Rightarrow$ struct
RETURN $\Rightarrow$ return
IF $\Rightarrow$ if
ELSE $\Rightarrow$ else
BREAK $\Rightarrow$ break
CONT $\Rightarrow$ continue
FOR $\Rightarrow$ for

2.2 Operators

Operators in Small-C are shown in the following table.

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
	$\sim$		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

[4] See section 2.2.

	<=		Not greater than
7	==		Equal to
	!=		Not equal to
8	&		Bit AND
9	^		Bit XOR
10	$\vert$		Bit OR
11	&&		Logical AND
12	$\vert\vert$		Logical OR
13	=	Right-to-left	Assign
	+=		+ and assign
	-=		- and assign
	*=		* and assign
	/=		/ and assign
	&=		& and assign
	^=		^ and assign
	$\vert$=		$\vert$ 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.

Implementation steps you may follow:

1. First, write the Lex program for Small-C , and store it in a *.l* file.

2. Second, run Flex compiler to compile the Lex program you have written. If everything goes fine, you will get a *.h* and a *.c* file having the same name with your Lex program.

3. Third, create a C/C++ project containing the previously generated *.h* and *.c* files, and compile the project to get your lexer.

4. Finally, test your lexer on several Small-C program samples. Your lexer should return a list of all the tokens in a program, with one token per line.

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 Step 2: Implementing a Syntax Analyzer

In this step, you will write a syntax analyzer to build a parse tree.

3.1 Grammar

PROGRAM	$\rightarrow$	EXTDEFS
EXTDEFS	$\rightarrow$	EXTDEF EXTDEFS
	$\mid$	ϵ
EXTDEF	$\rightarrow$	SPEC EXTVARS SEMI
	$\mid$	SPEC FUNC STMTBLOCK
EXTVARS	$\rightarrow$	DEC
	$\mid$	DEC COMMA EXTVARS
	$\mid$	ϵ
SPEC	$\rightarrow$	TYPE
	$\mid$	STSPEC
STSPEC	$\rightarrow$	STRUCT OPTTAG LC DEFS RC
	$\mid$	STRUCT ID
OPTTAG	$\rightarrow$	ID
	$\mid$	ϵ
VAR	$\rightarrow$	ID
	$\mid$	VAR LB INT RB
FUNC	$\rightarrow$	ID LP PARAS RP
PARAS	$\rightarrow$	PARA COMMA PARAS
	$\mid$	PARA
	$\mid$	ϵ
PARA	$\rightarrow$	SPEC VAR
STMTBLOCK	$\rightarrow$	LC DEFS STMTS RC
STMTS	$\rightarrow$	STMT STMTS
	$\mid$	ϵ
STMT	$\rightarrow$	EXP SEMI
	$\mid$	STMTBLOCK

	$\mid$	RETURN EXP SEMI
	$\mid$	IF LP EXP RP STMT ESTMT
	$\mid$	FOR LP EXP SEMI EXP SEMI EXP RP STMT
	$\mid$	CONT SEMI
	$\mid$	BREAK SEMI
ESTMT	$\rightarrow$	ELSE STMT
	$\mid$	ϵ
DEFS	$\rightarrow$	DEF DEFS
	$\mid$	ϵ
DEF	$\rightarrow$	SPEC DECS SEMI
DECS	$\rightarrow$	DEC COMMA DECS
	$\mid$	DEC
DEC	$\rightarrow$	VAR
	$\mid$	VAR ASSIGNOP INIT
INIT	$\rightarrow$	EXP
	$\mid$	LC ARGS RC
EXP	$\rightarrow$	EXP BINARYOP EXP
	$\mid$	UNARYOP EXP
	$\mid$	LP EXP RP
	$\mid$	ID LP ARGS RP
	$\mid$	ID ARRS
	$\mid$	EXP DOT ID
	$\mid$	INT
	$\mid$	ϵ
ARRS	$\rightarrow$	LB EXP RB ARRS
	$\mid$	ϵ
ARGS	$\rightarrow$	EXP COMMA ARGS
	$\mid$	EXP

To make the project easier, we list some explanations and restrictions as follows.

- The meaning of statement in Small-C is based on the meaning in *C*.
- 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 are used.
- *Struct* can only contain *int* variables.
- The return type of a function can only be *int*.

3.2 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.

Implementation steps you may follow:

1. First, write the Yacc program for Small-C , and store it in a *.y* file.
2. Second, run Parser Generator to compile the Yacc program you have written. If everything goes fine, you will get a *.h* and a *.c* file having the same name with your Yacc program.
3. Third, create a C/C++ project containing the previously generated *.h* and *.c* files, and compile the project to get your syntax analyzer.
4. Finally, test your syntax analyzer on several Small-C program samples. Your syntax analyzer should return a parse tree, in which each node for a subtree should be indented under its parent node. The display of identifiers and numbers must include the semantic content of the node.

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/.

Requirements

1. Pack the source files into a file named StudentID.tar. Use meaningful names for the files, so that the contents of the file are obvious. A single makefile that makes the executables out of all the source codes should be provided in the submission. Enclose a README file that lists the files you have submitted along with a one sentence explanation.
2. Your analyzer will be tested by the following command:
./program "Source file name" "Output file name".
Your analyzer needs to read source code from the source file, and outputs the results to the output file. If there is any error, output "Error." to the output file. Please output other information to *stderr*.

3. Please state clearly the purpose of each program at the start of the source program, and clearly comment your programs.

4. Send your StudentID.tar file to sjtucs215@163.com.

5. Due date: Nov. 8, 2015, midnight.