Amirkabir University of Technology

Optional Project

SAYEH Extension
(C Compiler)

Siavash Kavousi
Farzan Dehbashi
Amihossein Sohrabbeig
Mohammadmahdi Samiipaghaleh

supervised by
Dr. Saeid Shiri Ghidari

May 27, 2017

Contents

1 Objective

A compiler is a computer program that transforms source code written in a high-level programming language (here C) into a lower level language (SAYEH Assembly).

This compiler consists the following phases:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Generation

2 Input Code

Input code consists of a simple C code within some constraints. like the following notes:

1. No scope of any variable is defined in this code.
2. This compiler doesn't support call of the functions.
3. Declare variables before writing any other statements.
4. There is " "(space) symbol between any token.
5. All different kinds of allowed keywords are discussed in section 4.2.
6. Input C contains nested statements like "if" and "while"

3 Lexical Analyzer

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The *Lexical Analyzer* breaks these syntaxes into a series of *Tokens*, by removing any whitespace or comments in the source code (just defined by "//" or "/* */").

If the *Lexical Analyzer* finds a token invalid, it generates an error (it specifies line of error, desired token and seen token). The *Lexical analyzer* works closely with the *Syntax Analyzer*. In case of termination of lexical process it will pass analyzed code to *Syntax Analyzer*.

3.1 Tokens

Lexemes(words) are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. tokens should be saved into two tables, *Symbol table* and it's specific *Token Table*, which will be discussed in Specification of Token Table section. *Symbol Table* has two columns:

1. **Token Column:** specifies token attribute (for example if token is **int**, this will be **keyword** or if token is **12**, this will be **number**)
2. **Index Column:** specifies token index in it's specific table (for example if we have token=keyword and index=11 in *Symbol Table*, it means what we want is in index 11 of keyword-table)

Then you save token details in it's specific table. Let's see the following example to get an intuition.

Example1. Assume that input is like this:

```
int value = 154;
```

The output of lexer would be:

```
int (keyword)
value (identifier)
= (operator)
154 (number)
; (punctuation)
```

which can be saved like the following structure.

token	index
keyword	0
identifier	0
operator	0
number	0
punctuation	0

Table 1: symbol-table

index	type
0	int

Table 2: keyword-table

In this example lexer reads every lexemes and adds it's metadata to related tables.
Note: Don't worry about register addr and memory addr columns in tables, they gonna be used in an upcomming phase.

index	value	register addr	memory addr
0	value	null	null

Table 3: identifier-table

index	name	value	register address	memory address
0	s0	154	null	null

Table 4: number-table

index	value
0	=

Table 5: operator-table

index	value
0	;

Table 6: punctuation-table

3.2 Specification of Tokens

1. **Keywords**
   (a) if
   (b) else
   (c) while
   (d) int
   (e) char
   (f) bool
   (g) null
   (h) true
   (i) false
2. **Operators**
   (a) **Assignment Operators**
      i. **= :** Simple assignment operator. Assigns values from right side operands to left side operand.
      ii. **+= :** Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.
      iii. **-= :** Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.

iv. ***= :** Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.

v. **/= :** Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.

vi. **%= :** Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.

(b) **Logical Operators**

i. **&& :** Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.

ii. **|| :** Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.

iii. **Exclamation Mark :** Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

(c) **Relational Operators**

i. **== :** Checks if the values of two operands are equal or not. If yes, then the condition becomes true.

ii. **!= :** Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.

iii. **> :** Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.

iv. **< :** Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.

v. **>= :** Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.

vi. **<= :** Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.

(d) **Arithmetic Operators**

i. **+ :** Adds two operands.

ii. **- :** Subtracts two operands.

iii. *** :** Multiplies both operands. (we assume one of the operands is 2 or a power of 2)

iv. **/ :** Divides numerator by denominator (we assume that denominator is 2 or a power of 2).

v. **++ :** Increment operator increases the integer value by one.

vi. **−− :** Decrements operator decreases the integer value by one.

3. **Identifiers:** Identifiers should follow this rule, otherwise lexer would announce an error:
   $\wedge([a-z][A-Z]) + ([a-z][A-Z][0-9])*$
   It means identifiers should start with letters.
4. **Numbers** This compiler supports negative and positive integers like:
   (a) 269
   (b) -278
   (c) +212
5. **Characters**
   (a) 'u'
6. **Punctuation marks**
   (a) ,
   (b) (
   (c))
   (d) }
   (e) {
   (f) : (colon)
   (g) ; (semicolon)

3.3 Longest Match Rule

When the *Lexical Analyzer* read the source-code, it scans the code letter by letter; and when it encounters a white-space, operator-symbol, or special-symbols, it decides that a word is completed.

3.4 Important Notes

- For simplicity all tokens are separated within a " "(space) character.
- In case of defining an *id*, if it hasn't got any value it's value would be "unknown".
- There are some " ", "/n", "/t" characters in the input code, and all these should be handled and filtered if needed by Compiler.
- Identifiers types and values are considered in Syntax Analyzer and Semantic Analyzer.
- *Syntax Analyzer* will catch errors and report all these elements:
  - C source code line in which error has occurred.
  - Desired *Token*
  - Seen *Token*

4 Syntax Analyzer

Generally **Syntax Analyzer** or **Parser** takes the input from a *Lexical Analyzer* in the form of token streams. The *Parser* analyzes the source code (token stream) against the production rules to detect any errors in the code (actually in real world, not in this project! :D). But here you can implement it simpler than that, because; what we want from this compiler, is a limited number of statements.

One section of the *Syntax Analyzer* is handling expressions. to fulfill this, it's better to use *Postfix* and *Prefix* expressions and using C stack to handle the expressions, just in the way you used to in *Data Structure* Course.

type	grammer	example
variable declarations	keyword id; keyword id, id; keyword id = expression;	int a; float a, b; int a = 3*4+5;

assignments and statements	id = expression
	id *= expression
	id /= expression
	id (operators defined above) expression
		a = b + 3+9*6;
		a *= b;
		a /= b;
		...

conditions	expression == expression expression != expression expression && expression expression || expression	a == b*2 a != 3*4 a && b a || b
if expressions	if (conditions) expressions if (conditions) expressions else expressions	if (a==b*2) int z=a; if (a==b*2) int z=a; else int z=b;
while expressions	while (conditions) expressions	while (a==b*2) int z=a;

Table 7: Grammers

4.1 Symbol Table

Symbol Table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. *Symbol Table* is used by both, the analysis and the synthesis parts of a compiler.

It maintains an entry for each name in the following format:

$$symbolname, type, attribute$$

For example, if a symbol table has to store information about the following variable declaration:
static int interest;
then it should store the entry such as:
$interest, int, static$
The attribute clause contains the entries related to the name.

4.2 Important Notes

- Any implementation of *Symbol Table* is acceptable.
- Pay attention to section 7.1(Register file and RAM Cells Management) for maintaining the values and being aware of their current place.

5 Semantic Analysis (Optional)

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not. main errors that is desired to be caught are the following:

1. **Division by Zero**
   - int Siavash = 0 ;
   - int b = 10 / Siavash ;
2. **Type Checking**
   - Char c = 2 ;
3. **Value Checking (Undefined Value)**
   - int a ;
   - int b = a - 2 ;
4. **Undeclared variable**
   - a = 2 ;
5. **Multiple Declaration of Variable**
   - int a = 2 ;
   - int a ;

6 Code generation

Code generation can be considered as the final phase of compilation. Generally in code generation phase we should consider two things:

- carrying the exact meaning of source code
- efficient in terms of cpu usage and memory management

Again here the latter one does not matter. In this phase you will transform source code directly to *SAYEH* assembly code. As we have a limited number of statements, we can easily generate code for them without using complex methods.

6.1 Register file and RAM Cells Management

One of the most important sections of this project is Management of the *Main Memory* and also *Register File*. Assume a scenario in which numerous registers are needed to perform the desired operation of a single expression and you are about to allocate several registers, and it leeds to a situation in which the whole registerfile is full. then compiler should store useless values stored in the *Register File* to *Main Memory* and load them again when needed to *Register File*.

To implement all these refer to *Symbol Table* section and Memory Address and Registerfile Address fields in the related tables.

6.2 Main Steps

- Generate code for each part separately
- Looking at *Symbol* table and token specific tables

example:

```
int a = 4;
while(a == 6) {
    a++;
}
```

We will break apart this example and generate code for each part based on symbols tables and *Grammers* table. We will also print necessary *Symbol Table* to have a better understanding of what's going on under the hood.

symbols table initial state

addr	type

Table 8: regmem-table

index	type
0	int
1	while

Table 9: keyword-table

index	value	register addr	memory addr
0	a	null	null

Table 10: identifier-table

variable declaration and assignment

As you see identifier **a** is on index 0 of identifier-table and **4** is on index 0 of

token	index
keyword	0
identifier	0
operator	0
number	0
punctuation	0
keyword	1
punctuation	1
identifier	1
operator	1
number	1
punctuation	2
punctuation	3
identifier	2
operator	2
punctuation	4
punctuation	5

Table 11: symbol-table

index	name	value	register address	memory address
0	s0	4	null	null
0	s1	4	null	null

Table 12: number-table

index	value
0	=
1	==
2	++

Table 13: operator-table

index	value
0	;
1	(
2	)
3	{
4	;
5	}

Table 14: punctuation-table

number-table. First you should set number (immediate) in a register then set that register in index 0 of identifier table which is index of **a** identifier.

```
mil R_01, low(t0)
```

```
mih R_0h, high(t0)
```

while expression

```
    jpa label2
label1:
    mil R_1l, low(1)
    mih R_1h, high(1)
    add R_0, R_1
label2:
    czf
    ccf
      while condition (you can generate code
      for this part separately)
      check brz and/or brc in order to branch
```

while condition As we talked about it before while condition can be any of the statements in while condition section of *Statement* table.

```
mil R_2l, low(t1)
mil R_2h, high(t1)
cmp R0, R2
brz label1
```

symbols table initial state

token	index
keyword	0
identifier	0
operator	0
number	0
punctuation	0
keyword	1
punctuation	1
identifier	1
operator	1
number	1
punctuation	2
punctuation	3
identifier	2
operator	2
punctuation	4
punctuation	5

Table 15: symbol-table

addr	type
0	r
1	r
2	r

Table 16: regmem-table

index	type
0	int
1	while

Table 17: keyword-table

index	value	register addr	memory addr
0	a	R0	null

Table 18: identifier-table

index	name	value	register address	memory address
0	s0	4	R0	null
0	s1	4	R2	null

Table 19: number-table

index	value
0	=
1	==
2	++

Table 20: operator-table

index	value
0	;
1	(
2	)
3	{
4	;
5	}

Table 21: punctuation-table

6.3 Important Notes

- Each variable is declared once, and it's name is just stored in it's proper symbol-table, so it is only known for your compiler not SAYEH. So SAYEH will never know the name of that variable, it would be fed by some assembly instructions to perform operations on the value stored in that proper

row of *Register File* or *Main Memory*.

7 Desired Output

Output desired for this section is a SAYEH assembly code saved to a file to be loaded by your SAYEH basic computer and run to catch the desired results. finally this result, and your registerfile values and Memory values would be graded.

7.1 Important Notes

- Real world compilers first transform source code to an intermediate representation (machine independent) then to assembly code (machine dependent).

8 Handin Instructions

1. Language of implementation of the compiler is optional but the only language is supported by TA team is Java.
2. Your output should be stored in a file and loaded by your VHDL SAYEH code and this project **includes VHDL code, needed for loading your generated code in SAYEH** and you will be graded by running C codes on *SAYEH Basic Computer*)
3. About 10% of your score would be devoted to the report you deliver within your projects code. In this report you would explain the whole job and anything special you have done. (note that it shouldn't be shorter than this document and should be typed!)
4. In addition to bonus sections mentioned beforehand, implementation of the whole SAYEH project (Cache, SAYEH Basic Computer) on *Altera DE2* FPGA boards will result in extra mark as well (Remember to change the RAM code to an synthesizable one).
5. All questions would be answered by the course's email: computerarchitecture95@gmail.com.
6. Place all your code, report and stuff into a single .zip file named as "FirstName LastName StudentID" before upload.
7. Deadline of the project is up to the last day of regular classes before the exam preparation week for any change contact the mentioned email.
8. In case of delivery, your code will be downloaded by the responsible TA from Moodle, so the only way to convey your code is Moodle and in if you need to reform your code please upload it when possible to be used in the due date.

9. **Cheating Alert!**

   (a) All your codes would be processed, both manually and automatically and in case of any similarity by any means, both of individuals involved, would be fined by getting -35 percent of the project's score (note that if pushing your code to Github or any other VCS, exposing your code to a friend or ... results in unexpected similarities of others, you ALL, are responsible!).

   (b) Any cooperation beyond members of the group is forbidden!

   (c) The only source you are allowed to copy form, is AUT/CEIT/Arch101 repo which has been devoted to this course, copying any other source from the Internet,... would be consider just like from another classmate.

   (d) By the way, in case of any similarity with any of the previous students of this course, you blame the same.

10. Remember that the last phase of the project is the most important one and includes about half of the whole project score.

Good Luck