EECS 3301 Project 1

Group 1

Version A

Jun Lin Chen

Student ID: **214533111**
CSE Account: **chen256**
Email: **chen256@my.yorku.ca**

Table of Contents

Instructions

To compile this program, you can use **`make`** command, or simply use **`cc s4.c -o s4.out`** to compile this program. This program keeps accepting input data from standard input until it receives EOF. To use this program, you can stream a file to standard input.

```
./s4.out < YOURCODEFILE.txt
```

For example:

```
./s4.out < sample-input.txt
```

Or, you can also run the program:

```
./s4.out
```

Type all your code to console, and then press **Ctrl+D** to send **EOF**.

Syntax of the code should be:

```
<letter> -> A | B | … | Z | a | b | … | z
<digit> -> 0 | 1 | 2 | 3 | 4 | 5| 6 | 7 | 8 | 9
<int_constant> -> <digit> { <digit> }
<id> -> <letter> { <letter> | <digit> }

<factor> -> id | int_constant | (<expr>)
<term> -> <factor> { * <factor>}
<expr> -> <term> { ( + | - ) <term> }

<s_print> -> print <expr>
<s_goto> -> goto <id>
<s_ifpos> -> ifpos <expr> goto <id>
<s_assign> -> <id> = <expr>
<s_label> -> label <id>
<s_comment> -> COMMENT:<anything except semicolon>

<statement> -> <s_print> |<s_assign> |<s_label> |<s_goto> |<s_ifpos> |<s_comment>
<program> -> <statement> { {;} <statement> } {[;]}
```

Design and Implementation

The most difficult thing is that the professor prevents me to use C++. Indeed, we do not need those object-oriented features from C++ for this program. However, compared with pure C approach, string operations and memory management operations are very convenience in C++. And we are not allowed to separate the C code file. Therefore, there are approximately 600 lines of code in one single file. We tried our best to write as much as comments in that file to make it easy to read.

And it is known to all that there is no string object in C. And string operations are very sophisticated in C. Although we have **`string.h`**, in this project, we rarely use string operations. However, we use pointer a lot. Because this is one of the advantage of C programming language.

Lexical Analysis

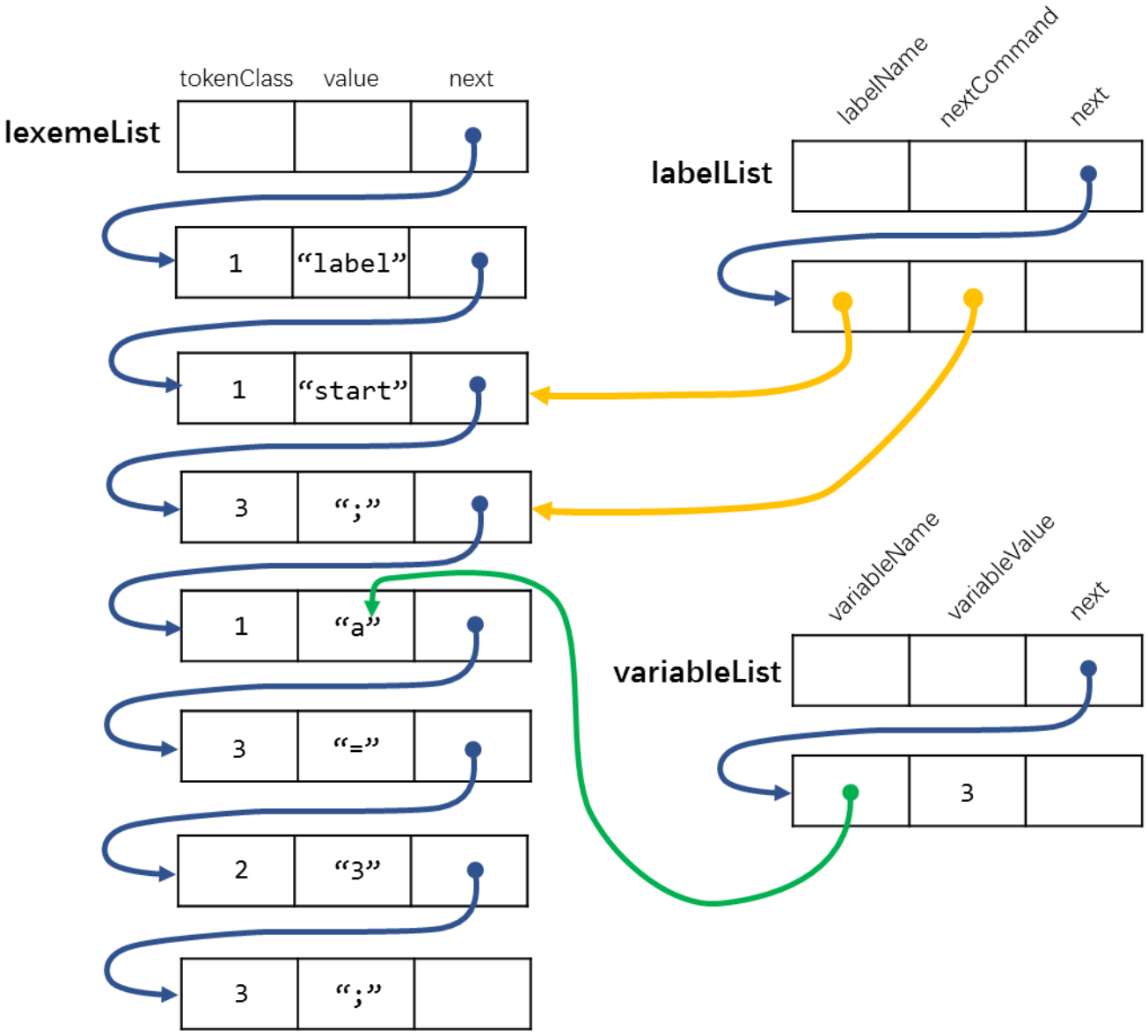
In the example given by the text book, it opens a file to do the lexical analysis. But in this implementation, we are using standard input. Considering that we need to parse the code twice later on, one for all the labels and one for computing the result, we need to somehow retain the code in the memory.

To avoid unnecessary string operations and make our code more elegant, we tokenize the code first. And then we use a linked list to store all the lexemes as lexeme structures. And we also set the **`MAX_LEXEME_LENGTH`** to 100. So, all the lexemes can only have maximum 100 characters.

While doing lexical analysis, the analyser also checks the type of the token and save it to the lexeme structure for convenience.

And then, all the variable structures and label structures can have their pointer pointed to the lexeme structure. We do not need to allocate new memory for storing the variable/label name when we add new variable/label to variable/or label list. Because we can point the pointer in variable/label structure to the corresponding lexeme structure. We also do not know how many lexemes in the code. In my opinion, using linked list is the best approach. Because linked list be easily extended.

The data structure for this program looks like this graph (on next page).



Variables and Labels

We also use linked list to store all the variables and labels. The reason why we would like to use linked list is that we can easily increase the number of the variables and labels. As the graph shown above, variables and labels are stored in different list. So, one **<id>** can be defined as a variable and a label in the same program.

For label structure, it has three pointers:

1. **labelName**: points to the name of the label.
2. **nextCommand**: points to the end of the label statement. So, the interpreter knows where it should jump to.
3. **next**: for the linked list data structure, pointing to the next element.

For the variable structure, it has two pointers and one long integer:

1. **`variableName`**: points to the name of the variable.
2. **`variableValue`**: stores the value of that variable.
3. **`next`**: for the linked list data structure, pointing to the next element.

Label Parsing

The **`goto`** statement should be able to not only jump backward but also jump forward. To achieve this goal, the interpreter will first go through all the code to mark all the labels. Then, when the interpreter is interpreting a **`goto`** statement (in the second parsing), it already has a list of all the labels. If interpreter can find that label in the label list. It returns the pointer that points to the end of the corresponding **`label`** statement. That means the interpreter will start interpreting from the end of that **`label`** statement.

In second parsing, the parser will just ignore all the **`label`** statements.

Recursive-Descent Parsing

The parsing reminds me the depth first search that RDP functions are called recursively. And they are finding the right parsing path. We come across some problems. To be notice that this interpreter is case-sensitive.

Double-Semicolons Issue

If we follow the syntax given by the project requirement to generate code, one statement should have at least two semicolons. We have discussed with our professor. He said one semicolon is good enough to end a statement. Therefore, we decided to check the semicolon in the syntax of **`<program>`**. The semicolons in the syntax of **`<s_print>`**, **`<s_assign>`**, **`<s_label>`**, **`<s_goto>`** and **`<s_ifpos>`** are removed.

But in **`<s_goto>`** and **`<s_ifpos>`**, semicolon will still be inspected inside the statements' RDP function. Because the interpreter may jump to other statement immediately.

Soft / Hard Mode Design

In our implementation, keyword (**`print label goto ifpos`**) can be used as label name and variable name. sometimes it may have multiple choices for one RDP function. RDP functions should be able to provide the best fit parse tree and it should have a trace back feature.

Therefore, we decide to make one RDP function into two parts. One is the soft part, the other is the hard part. In “SOFT”, if the RDP function cannot accept the statement, it will just return to its caller doing anything. In “HARD”, if the RDP function cannot accept the statement, it will throw a syntax error and terminate the program.

Extra Feature

We also add a COMMENT feature in our interpreter. To make it easy, we use as similar structure to other statements:

`<s_comment> -> COMMENT:<anything except semicolon>`

Testcases

sample-input.txt

This sample input is shared by all the group members. It examines all the operations.

```
COMMENT: EECS 3301 Project 1 Version A;
COMMENT:=====================================================;
COMMENT:    Jun Lin Chen    chen256@my.yorku.ca;
COMMENT:    Vishal Malik    vishal27@my.yorku.ca;
COMMENT:    Tong Wu         malan52.82@gmail.com;
COMMENT:=====================================================;
COMMENT: This is a public test file shared by all group member.;


COMMENT: Compute and print basic expression;
print (2 - 1) * 6 + 5;

COMMENT: Assign variables;
a = (3+22);
b = a - 3;
print b;

COMMENT: Skip statements using goto label;
c = 33;
goto tag1;
c = c - 1;
label tag1;
print c;

COMMENT: Conditional loops using ifpos;
d = 0;
label tag2;
d = d + 1;
print d;
ifpos 4-d goto tag2;

COMMENT: Special cases;
label tag3;;;;
print = 54;
print print + 1
```

Output:

```
11
22
33                  Jumping forward. Interpreter ignores the statement "c = c - 1;"
1                   Counting in loop.
2
3
4
55                  Keyword can be variable name
```

sample-input-01.txt

This testcase shows that extra spaces or lines do not matter in the interpretation. Because the code will be tokenized by the lexical analyzer.

```
COMMENT: EECS 3301 Project 1 Version A;
COMMENT: Extra spaces or lines do not matter;
print 1;        a=   0-10   +      3*5;
label backToHere;
   print a;a=a-1;
ifpos a goto backToHere;

print
233333;
```

Output:

```
1
5
4
3
2
1
233333
```

sample-input-02.txt

This testcase shows how to embed loops.

```
COMMENT: EECS 3301 Project 1 Version A;
COMMENT: embed loops;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
i=0;label LOOP1;
   i=i+1;ifpos i-3 goto ExitLoop1;

   j=0; label LOOP2;
      j=j+1;ifpos j-3 goto ExitLoop2;
      print i*10+j;

   goto LOOP2;
   label ExitLoop2;

goto LOOP1;
label ExitLoop1;
```

Output:

```
11
12
13
21
22
23
31
32
33
```

sample-input-03.txt

Input:

```
a=2;
print a ++ 3;
```

Output:

```
Error:
       In <factor> lexeme "+".
       Syntax: CANNOT Find a valid 'id | int_constant | (<expr>)'
```

In this case, the interpreter is hanging in the parse path:

<program> -> <statement> -> <s_print> -> <expr> -> <term> -> <factor>.

The reading pointer is pointing to the second plus sign.

sample-input-04.txt

Input:

```
a=2;
print a  3;
```

Output:

```
2
Error:
       In <program> lexeme "3".
       Syntax: Statement CANNOT Be Resolved.
```

In this case, the interpreter thinks that "**`print a`**" is a valid statement. Therefore, it prints "2". However, the program is hanging in **<program>**. Because this statement ("**`print  a`**") is not terminated by a semicolon before the next statement.

sample-input-05.txt

Input:

```
print 1; goto tag1;
label tag1; print 2;
label tag1; print 3;
```

Output:

```
1
3
```

Label can be re-declared. The latest label will over-write the old one.

sample-input-06.txt

Input:

```
a = a +3;
```

Output:

```
Error:
       In variable searching process, for lexeme "a".
       Variable Not Defined
```

Variable should be assigned before using.

sample-input-07.txt

Input:

```
a = (1 + 10 * 2) - ( 2* (5 + 5));
print (a);
b = ((3 + 2);
```

Output:

```
1
Error:
       In <factor> lexeme ";".
       Syntax: Expecting Right Parenthesis for '(<expr>).'
```

In the first line and second line, parentheses are matching. So, the interpreter can calculate the result. However, in line three, the right parenthesis is missing. The interpreter is halt at:

<factor> -> id | int_constant | (<expr>)

Answers for the Questions

#5.1 & #5.2

This is very simple. Similar to other RDP functions. I have implement the **`set`** statement in **s4upd.c**.

- Advantage: Easy to implement in RDP. No need to use the "HARD"/"SOFT" mode design that I mention before.
- Disadvantage: Readability decreases.

#5.3

Interpreter can search for the **`label`** statement after reached **`goto`** statement. But this approach makes the parsing procedure too sophisticated. That is the reason why we decide to run a separate parsing for the labels. There will be two parsing in our interpreter.

#5.4 & #5.5

No, we need more information before implementing this kind of statement. So, I do not have code for this statement.

Inside the **`if`** statement, statement itself can embed another **`if`** statement. And this will cause ambiguous. RDP parser cannot properly resolve the **`end`** keyword for the first **`if`** statement.

```
if <boolexpr> then
if <boolexpr> then
    <statement>
end;
else
    <statement>
end ;
```

The syntax **`<s_if> -> if <boolexpr> then <statement> [else <statement>] end ;`** is problematic.

Therefore, we may need definition for the code block. We may need to use new keywords **begin** and **end**. For example, I think I would like to add these syntax rules:

`<program> -> <block> { {;} <block> } [;]`

`<block> -> begin <statement> { {;} <statement> } end`

`<s_if> -> if (<boolexpr>) then <block> [else <block>] end;`

And we do not have the syntax for **`<boolexpr>`**.

`<boolexpr> -> <expr> {( < | <= | == | != | >= | > ) <expr>}`