Testing Document

Lina Vo

CSC 450

April 25, 2016

1 Scanner

The following tests are used to ensure that the scanner component of the compiler is functioning properly.

1.0 Comments

1.0.1 Legal comment

The legal comment tested included keywords, digits, symbols that are legal and illegal in the language, and IDs. Nothing was output.

Input

```
{ this is for csc 450, hi i'm lina vo. here are some
symbols! *&@$*() integer }
```

Output

1.0.2 Illegal comment

The illegal comment included, '{' which is illegal in the language. View Lexical Conventions.

Input

```
{ { check out this invalid comment...  }
```

Output

```
Error! Invalid token { was found.
```

1.1 Keywords

1.1.1 Legal keywords

All of the legal keywords were tested to see if the scanner would recognize the proper token for each keyword. All of the tokens returned corresponded with their keywords.

Input

```
program div while or and mod if var integer array of real
function procedure begin end then else do not
```

Output

```
Found PROGRAM with lexeme program
Found DIV with lexeme div
Found WHILE with lexeme while
Found OR with lexeme or
Found AND with lexeme and
Found MOD with lexeme mod
Found IF with lexeme if
Found VAR with lexeme var
Found INTEGER with lexeme integer
Found ARRAY with lexeme array
Found OF with lexeme of
Found REAL with lexeme real
Found FUNCTION with lexeme function
Found PROCEDURE with lexeme procedure
Found BEGIN with lexeme begin
Found END with lexeme end
Found THEN with lexeme then
Found ELSE with lexeme else
Found DO with lexeme do
Found NOT with lexeme not
```

1.1.2 Keywords with digits

Tested a keyword with some digits. The scanner recognized it as an ID, not a keyword.

Input

```
program44
```

Output

```
Found ID with lexeme program44
```

1.1.3 Keyword start with other text

Tested a keyword with some other text. The scanner recognized it as an ID, not a keyword.

Input

```
programSTUFF
```

Output

```
Found ID with lexeme programSTUFF
```

1.1.4 Keyword with symbol at the end

Tested a keyword with a symbol at the end. The scanner recognized the keyword, then the symbol separately.

Input

```
procedure;
```

Output

```
Found PROCEDURE with lexeme procedure
Found SEMICOLON with lexeme ;
```

1.2 IDs

1.2.1 ID with just letters

Testing what should be considered an ID with no digits or symbols. The scanner recognized the IDs properly and separately.

Input

```
myBankAccount
```

Output

```
Found ID with lexeme myBankAccount
```

1.2.2 ID with letters and digits

Testing what should be considered an ID with just digits, no symbols. The scanner recognized the ID properly with the correct lexeme.

Input

```
program44
```

Output

```
Found ID with lexeme myBankAccount
```

1.2.3 ID with letters and symbols

Testing what should be considered IDs and symbols separately. The scanner recognized IDs and the symbol separately and properly.

Input

```
my*BankAccount
```

Output

```
Found ID with lexeme my
Found MULTIPLY with lexeme *
Found ID with lexeme BankAccount
```

1.2.4 Illegal ID

Testing to see if an error will occur for an invalid ID, more specifically an ID with a digit in front.

Input

```
5helio
```

Output

```
Error! Invalid token 5h was found.
```

1.3 Numbers

1.3.1 Whole integer

Testing what should be considered an integer token. The scanner recognized the following as an integer and a symbol separately.

Input

```
5;
```

Output

```
Found INTEGER with lexeme 5
Found SEMICOLON with lexeme ;
```

1.3.2 Fraction

Testing what should be recognized as a real number and should be considered a REAL token. The scanner recognized the following as a REAL token.

Input

```
1.22
```

Output

```
Found REAL with lexeme 1.22
```

1.3.3 Exponent

Testing what should be recognized as a real number and should be considered a REAL token. The scanner recognized the following as a REAL token with the proper lexeme.

Input

```
1E10
```

Output

```
Found REAL with lexeme 1E10
```

1.3.5 Fraction with exponent

Testing what should be recognized a real number and as a REAL token. The scanner recognized the following as a REAL token with the proper lexeme.

Input

```
5.201E10
```

Output

```
Found REAL with lexeme 5.20E10
```

1.3.6 Whole integer with exponent and sign

Testing what should be recognized as a real number and as a REAL token regardless of the sign used. The scanner recognized the following as a REAL tokens with the proper lexemes.

Input

```
1E-10       5E+10
```

Output

```
Found REAL with lexeme 1E-10
Found REAL with lexeme 5E+10
```

1.3.7 Fraction with exponent and sign

Testing what should be recognized as a real number and as a REAL token regardless of the sign used. The scanner recognized the following as a REAL tokens with the proper lexemes.

Input

```
1.20E-10    4.2E+6
```

Output

```
Found REAL with lexeme 1.20E-10
Found REAL with lexeme 4.2E+6
```

1.3.8 Illegal numbers

Testing what should be considered illegal numbers in the language. All cases shown below would lead to the ERROR state since they are considered invalid tokens.

Input

```
1.    8E    1E+
```

Output

```
Error! Invalid token 1. was found.
Error! Invalid token 8E was found.
Error! Invalid token 1E+ was found.
```

1.4 Symbols

1.4.1 Short Symbols

Testing what should be considered legal short symbols in the mini Pascal language. All of the symbols below had the proper tokens as well as lexemes.

Input

```
<     :     >     =     +     -     *     [     ]     (     )
,     .     /     ;
```

Output

```
Found LESS_THAN with lexeme <
Found COLON with lexeme :
Found GREATER_THAN with lexeme >
Found ASSIGN with lexeme =
Found PLUS with lexeme +
Found MINUS with lexeme -
Found MULTIPLY with lexeme *
Found OPEN_BRACKET with lexeme [
Found CLOSE_BRACKET with lexeme ]
Found OPEN_PAREN with lexeme (
Found CLOSE_PAREN with lexeme )
Found COMMA with lexeme ,
Found PERIOD with lexeme .
Found DIVIDE with lexeme /
Found SEMICOLON with lexeme ;
```

1.4.2 Symbols

Testing what should be considered legal symbols in the mini Pascal language. All of the symbols below had the proper tokens as well as lexemes.

Input

```
<=    <>    >=    :=
```

Output

```
Found LESS_THAN_EQUALS with lexeme <=
Found LESS_THAN_GREATER_THAN with lexeme <>
```

```
Found GREATER_THAN_EQUALS with lexeme >=
Found ASSIGN_OP with lexeme :=
```

1.4.3 Illegal Symbols

Testing what should be considered illegal symbols in the mini Pascal language. All of the symbols would reach the ERROR state.

Input

```
^     &     $     %     #     @     !     ~     |     \     ‘
“     ?
```

Output

```
Error! Invalid token ^ was found.
Error! Invalid token & was found.
Error! Invalid token $ was found.
Error! Invalid token % was found.
Error! Invalid token # was found.
Error! Invalid token @ was found.
Error! Invalid token ! was found.
Error! Invalid token ~ was found.
Error! Invalid token | was found.
Error! Invalid token \ was found.
Error! Invalid token ‘ was found.
Error! Invalid token “ was found.
Error! Invalid token ? was found.
```

1.5 Miscellaneous

1.5.1 Blank text file

Testing to see what the scanner does when you give it a blank file. The scanner exits properly with no error. No tokens are found.

1.5.2 End of file

Testing to see if the end of file values are correct. There should be no next token. The lexeme and token should be null.

1.5.3 File not found

Testing to see if the scanner will return a proper error message when the file is not found. The scanner returns a message saying that the file was not found.

Input

```
/Users/LinaVo/vo-compiler/input.txt
```

Output

```
Could not find file /Users/LinaVo/vo-compiler/input.txt
```

2 Recognizer

The following tests are used to ensure that the recognizer component of the compiler is functioning properly.

2.0.0 Legal Empty Program

Testing an empty mini pascal program.

Input

```
program foo;
begin
end
.
```

Output

2.0.1 Illegal Empty Program

Testing an illegal empty mini pascal program.

Input

```
program
begin foo
end
.
```

Output

```
An error occurred. Invalid BEGIN found.
```

2.0.2 Legal Simple Program

Testing a legal simple program that includes instantiation variables.

Input

```
program foo;
var helio : integer;
begin
if helio then fire else phew
end
.
```

Output

2.0.3 Illegal Simple Program

Testing an illegal simple program.

Input

```
program foo;
var helio
begin
if
end
.
```

Output

```
An error occurred. Invalid BEGIN found.
```

2.0.4 Legal Moderately Complex Program

Testing a legal moderately complex program.

Input

```
program foo;
var helio : real;
procedure fire;
begin
if helio then fire else phew
end;
begin
end
.
```

Output

2.0.5 Illegal Moderately Complex Program

Testing an illegal moderately complex program.

Input

```
program foo;
var helio : 24;
fire
procedure;
begin
end
.
```

Output

```
An error occurred. Invalid ID found.
```

2.0.6 Legal Complex Program

Testing a legal complex program.

Input

```
program foo;
var wow : integer;
function stars ( light : real ) : integer;
begin
while stars + wow do fire
end;
begin
end
.
```

Output

2.0.7 Illegal Complex Program

Testing an illegal complex program.

Input

```
program foo;
var wow : 19;
function stars
begin
while stars + wow do fire
end
.
```

Output

```
An error occurred. Invalid NUM found.
```

3 Recognizer + Symbol Table

The following tests are used to ensure that the recognizer component with the addition of the symbol table of the compiler is functioning properly.

3.0.0 Legal Empty Program

Testing an empty mini pascal program.

Input

```
program foo;
begin
end
.
```

Output

```
You have reached the end of the file.
--------------------------------------
Lexeme        Kind
--------------------------------------
foo           PROGRAM
--------------------------------------
```

3.0.1 Illegal Empty Program

Testing an illegal empty mini pascal program.

Input

```
program
begin foo
end
.
```

Output

```
An error occurred. Invalid BEGIN found.
--------------------------------------
Lexeme        Kind
--------------------------------------
--------------------------------------
```

3.0.2 Legal Simple Program

Testing a legal simple program that includes instantiation variables.

Input

```
program foo;
var helio, fire : integer;
begin
if foo then helio else fire
end
.
```

Output

```
You have reached the end of the file.
--------------------------------------
Lexeme        Kind
--------------------------------------
fire          VAR
helio         VAR
foo           PROGRAM
--------------------------------------
```

3.0.3 Illegal Simple Program

Testing an illegal simple program.

Input

```
program foo;
var foo
begin
if
end
.
```

Output

```
Error - foo already exists.
---------------------------------------
Lexeme        Kind
---------------------------------------
foo           PROGRAM
---------------------------------------
```

3.0.4 Legal Moderately Complex Program

Testing a legal moderately complex program.

Input

```
program foo;
var helio : real;
procedure fire;
begin
if helio then fire else foo
end;
begin
end
.
```

Output

```
You have reached the end of the file.
---------------------------------------
Lexeme        Kind
---------------------------------------
fire          PROCEDURE
helio         VAR
foo           PROGRAM
---------------------------------------
```

3.0.5 Illegal Moderately Complex Program

Testing an illegal moderately complex program.

Input

```
program foo;
var helio : real;
procedure fire;
begin
if helio then fire else phew
end;
begin
end
.
```

Output

```
Error - phew doesn't exist.
```

```
--------------------------------------
Lexeme        Kind
--------------------------------------
fire          PROCEDURE
helio         VAR
foo           PROGRAM
--------------------------------------
```

3.0.6 Legal Complex Program

Testing a legal complex program.

Input

```
program foo;
var wow : integer;
function stars ( light : real ) : integer;
begin
while stars + wow do foo
end;
begin
end
.
```

Output

```
You have reached the end of the file.
--------------------------------------
Lexeme        Kind
--------------------------------------
stars         FUNCTION
wow           VAR
light         VAR
foo           PROGRAM
--------------------------------------
```

3.0.7 Illegal Complex Program

Testing an illegal complex program.

Input

```
program foo;
var wow : integer;
function stars ( light : real ) : integer;
begin
while stars + wow do fire
end;
begin
end
.
```

Output

```
Error - fire doesn't exist.
--------------------------------------
Lexeme        Kind
--------------------------------------
stars         FUNCTION
wow           VAR
light         VAR
foo           PROGRAM
--------------------------------------
```

4 Syntax Tree

The following test is used to ensure that the integration of the syntax tree to the parser is covering all the necessary cases.

4.0.0 Bitcoin Example

The following example was the in class base case example for the syntax tree. Other tests were run to ensure that the other nodes were being printed out properly. This one is to demonstrate that it prints out the proper toString() for the base case.

Input

```
program sample;
var dollars, bitcoins, yen : integer;
begin
dollars := 1000000;
yen := dollars * 102;
bitcoins := dollars / 400;
end
.
```

Output

```
indented syntaxTree shown below
---------------------------------------

ProgramNode: sample
|-- DeclarationsNode
|-- -- ExpressionNode of type nullVariableNode: dollars
|-- -- ExpressionNode of type nullVariableNode: bitcoins
|-- -- ExpressionNode of type nullVariableNode: yen
|-- SubprogramDeclarationsNode
|-- CompoundStatementNode
|-- -- AssignmentStatementNode
|-- -- -- ExpressionNode of type nullVariableNode: dollars
|-- -- -- ExpressionNode of type nullValueNode: 1000000
|-- -- AssignmentStatementNode
|-- -- -- ExpressionNode of type nullVariableNode: yen
|-- -- -- ExpressionNode of type nullOperationNode: MULTIPLY
|-- -- -- -- ExpressionNode of type VARVariableNode: dollars
|-- -- -- -- ExpressionNode of type nullValueNode: 102
|-- -- AssignmentStatementNode
|-- -- -- ExpressionNode of type nullVariableNode: bitcoins
|-- -- -- ExpressionNode of type nullOperationNode: DIVIDE
|-- -- -- -- ExpressionNode of type VARVariableNode: dollars
|-- -- -- -- ExpressionNode of type nullValueNode: 400
```

5 Code Generation

The following tests are used to ensure that the code generation produces proper MIPS assembly code.

5.0.0 Bitcoin Example

Testing in class bitcoin example.

Input

```
program sample;
var dollars, bitcoins, yen : integer;
var watson : integer;
begin
    dollars := 1000000;
    yen := dollars * 102;
    bitcoins := dollars/400;
end
.
```

Output

```
.data
dollars: .word 0
bitcoins: .word 0
yen: .word 0
watson: .word 0

.text
main:
li $t0, 1000000
sw $t0, dollars
lw $t0, dollars
li $t1, 102
mult $t0, $t1
mflo $t0
sw $t0, yen
lw $t0, dollars
li $t1, 400
div $t0, $t1
mflo $t0
sw $t0, bitcoins
jr $ra
```

5.1.0 Operators - Addition, subtraction, multiplication, division

Testing basic math operators

Input

```
program math;
var foo : integer;
var bar : integer;
var six, two : integer;
begin
  foo := 5;
  bar := 10;
  foo := foo + bar;
  bar := 123 - 3;
  six := 2 * 3;
  two := 10 / 5;
```

```
end
.
```

Output

```
.data
foo: .word 0
bar: .word 0
six: .word 0
two: .word 0

.text
main:
li $t0, 5
sw $t0, foo
li $t0, 10
sw $t0, bar
lw $t0, foo
lw $t1, bar
add $t0, $t1, $t0
sw $t0, foo
li $t0, 123
li $t1, 3
sub $t0, $t0, $t1
sw $t0, bar
li $t0, 2
li $t1, 3
mult $t0, $t1
mflo $t0
sw $t0, six
li $t0, 10
li $t1, 5
div $t0, $t1
mflo $t0
sw $t0, two
jr $ra
```

5.1.3 Operators - Equals

Testing to see if values are equal. This case they are not. I also checked if they were.

Input

```
program test;
var foo : integer;
begin
if 10  = 5
then foo := 1
else foo := 2
end
.
```

Output

```
.data
foo: .word 0

.text
main:
li $t0, 10
li $t1, 5
beq $t1, $t0, equalsOne
li $t1, 0
```

```
j endEqualsOne
equalsOne:
li $t1, 1
endEqualsOne:
beq $t1, $zero, ElseNum1
li $t0, 1
sw $t0, foo
j EndNum1
ElseNum1:
li $t0, 2
sw $t0, foo
EndNum1:
jr $ra
```

5.1.4 Operators - Less than or equal to

The example below is for when the value is equal to the number being checked. Testing was also completed on if the number was less than and if it was greater than.

Input

```
program test;
var foo, bar : integer;
begin
foo := 5;
if foo <= 5
then foo := 15
else foo := 22
end
.
```

Output

```
.data
foo: .word 0
bar: .word 0

.text
main:
li $t0, 5
sw $t0, foo
lw $t0, foo
li $t1, 5
sle $t1, $t0, $t1
beq $t1, $zero, ElseNum1
li $t0, 15
sw $t0, foo
j EndNum1
ElseNum1:
li $t0, 22
sw $t0, foo
EndNum1:
jr $ra
```

5.1.x Operators - Greater than

This example below is testing specifically to equate false. Testing for true also happened but is not listed here to save room.

Input

```
program test;
var number : integer;
```

```
begin
if 20 > 50
then number := 10
else number := 20
end
.
```

Output

```
.data
number: .word 0

.text
main:
li $t0, 20
li $t1, 50
sgt $t1, $t0, $t1
beq $t1, $zero, ElseNum1
li $t0, 10
sw $t0, number
j EndNum1
ElseNum1:
li $t0, 20
sw $t0, number
EndNum1:
jr $ra
```

5.1.5 Operators - Less than

The example below is testing specifically to equate true. Also, tested for false. Not listed in here.

Input

```
program test;
var number : integer;
begin
number := 5;
if number < 15
then number := 10
else number := 15
end
.
```

Output

```
.data
number: .word 0

.text
main:
li $t0, 5
sw $t0, number
lw $t0, number
li $t1, 15
slt $t1, $t0, $t1
beq $t1, $zero, ElseNum1
li $t0, 10
sw $t0, number
j EndNum1
ElseNum1:
li $t0, 15
sw $t0, number
```

```
EndNum1:
jr $ra
```

5.1.5 Operators - Greater than or equal to

Testing greater than or equal to. This case below is for a false value. I also tested if the number was greater and if it was equal. Both evaluate to true

Input

```
program test;
var foo : integer;
begin
foo := 5;
if 2 >= 5
then foo := 15
else foo := 22
end
.
```

Output

```
.data
foo: .word 0

.text
main:
li $t0, 5
sw $t0, foo
li $t0, 2
li $t1, 5
sge $t1, $t0, $t1
beq $t1, $zero, ElseNum1
li $t0, 15
sw $t0, foo
j EndNum1
ElseNum1:
li $t0, 22
sw $t0, foo
EndNum1:
jr $ra
```

5.1.6 Operators - Not equal

Testing not equal operator. This example is when the numbers are equal. Testing when the numbers are not was also done but is not listed here.

Input

```
program test;
var foo : integer;
begin
  foo := 10;
  if foo <> 10
  then foo := foo + 1
  else foo := foo - 5
end
.
```

Output

```
.data
foo: .word 0

.text
main:
```

```
li $t0, 10
sw $t0, foo
lw $t0, foo
li $t1, 10
beq $t1, $t0, equalsZero
li $t1, 0
j endEqualsZero
equalsZero:
li $t1, 0
endEqualsZero:
beq $t1, $zero, ElseNum1
lw $t0, foo
li $t1, 1
add $t0, $t1, $t0
sw $t0, foo
j EndNum1
ElseNum1:
lw $t0, foo
li $t1, 5
sub $t0, $t0, $t1
sw $t0, foo
EndNum1:
jr $ra
```

5.2.0 If Statements

Testing basic if statements and embedded if statements.

Input

```
program test;
var foo : integer;
var bar : integer;
begin
foo := 15;
if foo = 6
then
      if foo = 0
      then foo := 200
      else bar := 20
else foo := 20
end
.
```

Output

```
.data
foo: .word 0
bar: .word 0

.text
main:
li $t0, 15
sw $t0, foo
lw $t0, foo
li $t1, 6
beq $t1, $t0, equalsOne
li $t1, 0
j endEqualsOne
equalsOne:
li $t1, 1
```

```
endEqualsOne:
beq $t1, $zero, ElseNum1
lw $t0, foo
li $t1, 0
beq $t1, $t0, equalsOne
li $t1, 0
j endEqualsOne
equalsOne:
li $t1, 1
endEqualsOne:
beq $t1, $zero, ElseNum2
li $t0, 200
sw $t0, foo
j EndNum3
ElseNum3:
li $t0, 20
sw $t0, bar
EndNum3:
j EndNum2
ElseNum2:
li $t0, 20
sw $t0, foo
EndNum2:
jr $ra
```

5.3.0 While Statements

Testing basic while loop and embedded while loop.

Input

```
program sample;
var nui, wow : integer;
begin
nui := 1;
wow := 1;
while nui < 5
 do
 begin
 while wow < 3
do
wow := wow + 1;
nui := nui + 1;
end
end
.
```

Output

```
.data
nui: .word 0
wow: .word 0

.text
main:
li $t0, 1
sw $t0, nui
li $t0, 1
sw $t0, wow
WhileNum0:
lw $t0, nui
```

```
li $t1, 5
slt $t0, $t0, $t1
beq $t0, $zero, ExitNum0
WhileNum1:
lw $t0, wow
li $t1, 3
slt $t0, $t0, $t1
beq $t0, $zero, ExitNum1
lw $t0, wow
li $t1, 1
add $t0, $t1, $t0
sw $t0, wow
j WhileNum1
ExitNum1:
lw $t0, nui
li $t1, 1
add $t0, $t1, $t0
sw $t0, nui
j WhileNum0
ExitNum0:
jr $ra
```

5.4.0 Not Expressions - Operations, Values, Variables

Below is an example of testing a not on an expression (operation). I also tested it on values and variables successfully.

Input

```
program test;
var hello : integer;
begin
hello := 5;
if not hello = 5
then hello := 10
else hello := 20
end
.
```

Output

```
.data
hello: .word 0

.text
main:
li $t0, 5
sw $t0, hello
lw $t0, hello
li $t1, 5
beq $t1, $t0, equalsOne
li $t1, 0
j endEqualsOne
equalsOne:
li $t1, 1
endEqualsOne:
bne $t1, $zero, ElseNum3
li $t0, 10
sw $t0, hello
j EndNum3
ElseNum3:
```

```
li $t0, 20
sw $t0, hello
EndNum3:
jr $ra
```