Testing Results

Test 1: int

- The compiler takes in a simple additions source code piece of code and outputs the tokens as follows

Output: Source Code and Tokens

Output

Test program

```
{
        int a
        a = 4

        int b
        b = 2 + a
} $
```

Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_INT [ int ]

Line 2: T_ID [ a ]

Line 3: T_ID [ a ]

Line 3: T_SINGLE_EQUALS [ = ]

Line 3: T_DIGIT [ 4 ]

Line 5: T_INT [ int ]

Line 5: T_ID [ b ]

Line 6: T_ID [ b ]

Line 6: T_SINGLE_EQUALS [ = ]

Line 6: T_DIGIT [ 2 ]

Line 6: T_PLUS [ + ]

Line 6: T_ID [ a ]

Line 7: T_RBRACE [ } ]

Line 7: T_EOF [ $ ]
```

Log

Verbose Mode: OFF

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id a from line 2 into symbol table at scope: 0

Inserting id b from line 5 into symbol table at scope: 0

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Verbose Mode: On (Sample because it wouldn’t fit on one page)

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

T_LBRACE expected!

T_LBRACE consumed!

T_INT consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_INT consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_PLUS expected!
```

CST

CST

```
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Variable Declaration>
-----[int]
-----[a]
---<Statement List>
----<Statement>
-----<Assignment Statement>
------[a]
------[=]
------<Expression>
-------<Int Expression>
--------[4]
----<Statement List>
-----<Statement>
------<Variable Declaration>
-------[int]
-------[b]
-----<Statement List>
------<Statement>
-------<Assignment Statement>
--------[b]
--------[=]
--------<Expression>
---------<Int Expression>
----------[2]
----------[+]
----------<Expression>
-----------[a]
------[Statement List]
--[}]
-[$]
```

AST

AST

```
<BLOCK>
-<Variable Declaration>
--[int]
--[a]
-<Assignment Statement>
--[a]
--[4]
-<Variable Declaration>
--[int]
--[b]
-<Assignment Statement>
--[b]
--<Add>
---[2]
---[a]
```

Symbol Table

Symbol Table

Name	Type	Scope	Line #
a	int	0	2
b	int	0	5

.

Test 2: String

- The compiler takes in a string and produces the tokens as follows

Tokens

```
Line 1: T_LBRACE [ { ]
Line 2: T_INT [ int ]
Line 2: T_ID [ a ]
Line 3: T_ID [ a ]
Line 3: T_SINGLE_EQUALS [ = ]
Line 3: T_DIGIT [ 4 ]
Line 4: T_IF [ if ]
Line 4: T_LPAREN [ ( ]
Line 4: T_ID [ a ]
Line 4: T_DOUBLE_EQUALS [ == ]
Line 4: T_DIGIT [ 4 ]
Line 4: T_RPAREN [ ) ]
Line 4: T_LBRACE [ { ]
Line 5: T_PRINT [ print ]
Line 5: T_LPAREN [ ( ]
Line 5: T_QUOTE [ " ]
Line 5: T_ID [ h ]
Line 5: T_ID [ e ]
Line 5: T_ID [ l ]
Line 5: T_ID [ l ]
Line 5: T_ID [ o ]
Line 5: T_WHITE_SPACE [   ]
Line 5: T_ID [ w ]
Line 5: T_ID [ o ]
Line 5: T_ID [ r ]
Line 5: T_ID [ l ]
Line 5: T_ID [ d ]
Line 5: T_QUOTE [ " ]
Line 5: T_RPAREN [ ) ]
Line 6: T_RBRACE [ } ]
```

Verbose Mode: Off

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id a from line 2 into symbol table at scope: 0

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Verbose Mode: On

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

T_LBRACE expected!

T_LBRACE consumed!

T_INT consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_IF expected!

T_IF consumed!

T_LPAREN, T_TRUE or T_FALSE expected!

T_LPAREN consumed!

T_ID expected!

T_ID consumed!

T_DOUBLE_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_RPAREN consumed!

T_LBRACE expected!
```

CST

CST

```
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Variable Declaration>
-----[int]
-----[a]
---<Statement List>
----<Statement>
-----<Assignment Statement>
------[a]
------[=]
------<Expression>
-------<Int Expression>
--------[4]
----<Statement List>
-----<Statement>
------<If Statement>
-------[if]
-------<Boolean Expression>
--------[(]
--------<Expression>
---------[a]
--------[==]
--------<Expression>
---------<Int Expression>
----------[4]
--------[)]
-------<Block>
--------[{]
--------<Statement List>
---------<Statement>
----------<Print Statement>
-----------[print]
-----------[(]
-----------<Expression>
------------<String Expression>
-------------["]
-------------<Char List>
--------------[h]
--------------[e]
--------------[l]
--------------[l]
--------------[o]
--------------[ ]
--------------[w]
--------------[o]
--------------[r]
--------------[l]
```

AST

AST

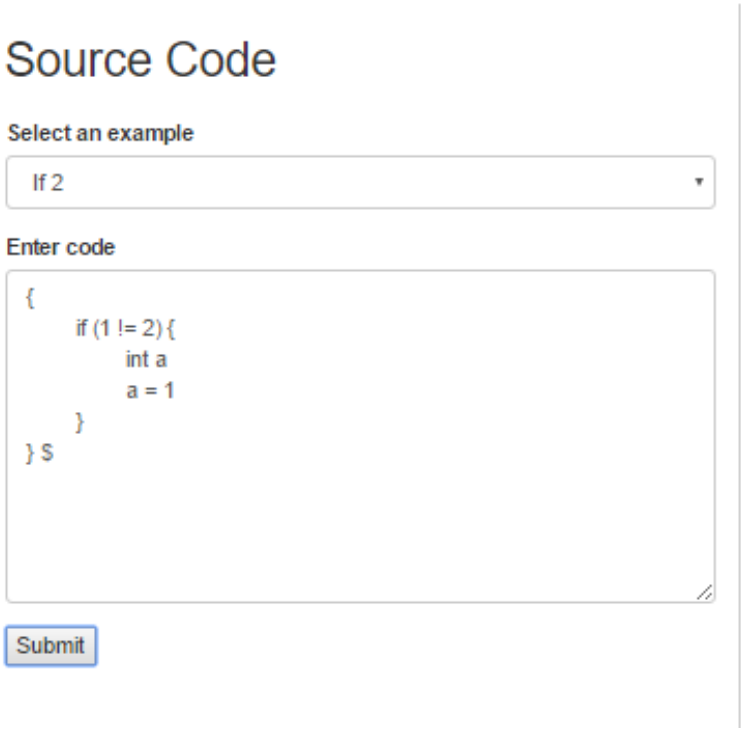
```
<BLOCK>
-<Variable Declaration>
--[int]
--[a]
-<Assignment Statement>
--[a]
--[4]
-<If Statement>
--<Equal>
---[a]
---[4]
--<BLOCK>
---<Print Statement>
----[hello world]
```

Symbol Table

Symbol Table

Name	Type	Scope	Line #
a	int	0	2

Test 3: if



- The compiler takes in source code that represents an if statement and returns all the tokens within the source code

Output

Test program

```
{
	if (1 != 2) {
		int a
		a = 1
	}
} $
```

Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_IF [ if ]

Line 2: T_LPAREN [ ( ]

Line 2: T_DIGIT [ 1 ]

Line 2: T_NOT_EQUALS [ != ]

Line 2: T_DIGIT [ 2 ]

Line 2: T_RPAREN [ ) ]

Line 2: T_LBRACE [ { ]

Line 3: T_INT [ int ]

Line 3: T_ID [ a ]

Line 4: T_ID [ a ]

Line 4: T_SINGLE_EQUALS [ = ]

Line 4: T_DIGIT [ 1 ]

Line 5: T_RBRACE [ } ]

Line 6: T_RBRACE [ } ]

Line 6: T_EOF [ $ ]
```

Verbose Mode:OFF

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id a from line 3 into symbol table at scope: 1

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Verbose Mode:ON

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

T_LBRACE expected!

T_LBRACE consumed!

T_IF expected!

T_IF consumed!

T_LPAREN, T_TRUE or T_FALSE expected!

T_LPAREN consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_NOT_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_RPAREN consumed!

T_LBRACE expected!

T_LBRACE consumed!

T_INT consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_DIGIT expected!
```

CST

```
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<If Statement>
-----[if]
-----<Boolean Expression>
------[(]
------<Expression>
-------<Int Expression>
--------[1]
------[!=]
------<Expression>
-------<Int Expression>
--------[2]
------[)]
-----<Block>
------[{]
------<Statement List>
-------<Statement>
--------<Variable Declaration>
---------[int]
---------[a]
-------<Statement List>
--------<Statement>
---------<Assignment Statement>
----------[a]
----------[=]
----------<Expression>
-----------<Int Expression>
------------[1]
--------[Statement List]
------[}]
---[Statement List]
--[}]
-[$]
```

AST

```
<BLOCK>
-<If Statement>
--<Not Equal>
---[1]
---[2]
--<BLOCK>
---<Variable Declaration>
----[int]
----[a]
---<Assignment Statement>
----[a]
----[1]
```

Symbol Table

Name	Type	Scope	Line #
a	int	1	3

Test 4: While

Source Code

Select an example

While

Enter code

```
{
	int x
	x = 0

	while (x != 5)
	{
		print(x)
		x = 1 + x
	}
} $
```

Submit

Output

Test program

```
{
	int x
	x = 0

	while (x != 5)
	{
		print(x)
		x = 1 + x
	}
} $
```

Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_INT [ int ]

Line 2: T_ID [ x ]

Line 3: T_ID [ x ]

Line 3: T_SINGLE_EQUALS [ = ]

Line 3: T_DIGIT [ 0 ]

Line 5: T_WHILE [ while ]

Line 5: T_LPAREN [ ( ]

Line 5: T_ID [ x ]

Line 5: T_NOT_EQUALS [ != ]

Line 5: T_DIGIT [ 5 ]

Line 5: T_RPAREN [ ) ]

Line 6: T_LBRACE [ { ]

Line 7: T_PRINT [ print ]

Line 7: T_LPAREN [ ( ]

Line 7: T_ID [ x ]

Line 7: T_RPAREN [ ) ]

Line 8: T_ID [ x ]

Line 8: T_SINGLE_EQUALS [ = ]

Line 8: T_DIGIT [ 1 ]

Line 8: T_PLUS [ + ]

Line 8: T_ID [ x ]

Line 9: T_RBRACE [ } ]

Line 10: T_RBRACE [ } ]

Line 10: T_EOF [ $ ]
```

Verbose mode : OFF

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id x from line 2 into symbol table at scope: 0

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Verbose Mode: On

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

T_LBRACE expected!

T_LBRACE consumed!

T_INT consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_WHILE expected!

T_WHILE consumed!

T_LPAREN, T_TRUE or T_FALSE expected!

T_LPAREN consumed!

T_ID expected!

T_ID consumed!

T_NOT_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_RPAREN consumed!
```

CST

```
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Variable Declaration>
-----[int]
-----[x]
---<Statement List>
----<Statement>
-----<Assignment Statement>
------[x]
------[=]
------<Expression>
-------<Int Expression>
--------[0]
----<Statement List>
-----<Statement>
------<While Statement>
-------[while]
-------<Boolean Expression>
--------[(]
--------<Expression>
---------[x]
--------[!=]
--------<Expression>
---------<Int Expression>
----------[5]
--------[)]
-------<Block>
--------[{]
--------<Statement List>
---------<Statement>
----------<Print Statement>
-----------[print]
-----------[(]
-----------<Expression>
------------[x]
-----------[)]
---------<Statement List>
----------<Statement>
-----------<Assignment Statement>
------------[x]
------------[=]
------------<Expression>
-------------<Int Expression>
--------------[1]
--------------[+]
--------------<Expression>
```

AST

```
<BLOCK>
-<Variable Declaration>
--[int]
--[x]
-<Assignment Statement>
--[x]
--[0]
-<While Statement>
--<Not Equal>
---[x]
---[5]
--<BLOCK>
---<Print Statement>
----[x]
---<Assignment Statement>
----[x]
----<Add>
-----[1]
-----[x]
```

Symbol Table

Name	Type	Scope	Line #
x	int	0	2

Test 5: Boolean

Source Code

Select an example

Boolean

Enter code

```
{
	int a
	a = 1

	boolean b
	b = (true == (true != (false == (true != (false != (a == a))))))

	print(b)
} $
```

Submit

Tokens

```
Line 1: T_LBRACE [ { ]
Line 2: T_INT [ int ]
Line 2: T_ID [ a ]
Line 3: T_ID [ a ]
Line 3: T_SINGLE_EQUALS [ = ]
Line 3: T_DIGIT [ 1 ]
Line 5: T_BOOLEAN [ boolean ]
Line 5: T_ID [ b ]
Line 6: T_ID [ b ]
Line 6: T_SINGLE_EQUALS [ = ]
Line 6: T_LPAREN [ ( ]
Line 6: T_TRUE [ true ]
Line 6: T_DOUBLE_EQUALS [ == ]
Line 6: T_LPAREN [ ( ]
Line 6: T_TRUE [ true ]
Line 6: T_NOT_EQUALS [ != ]
Line 6: T_LPAREN [ ( ]
Line 6: T_FALSE [ false ]
Line 6: T_DOUBLE_EQUALS [ == ]
Line 6: T_LPAREN [ ( ]
Line 6: T_TRUE [ true ]
Line 6: T_NOT_EQUALS [ != ]
Line 6: T_LPAREN [ ( ]
Line 6: T_FALSE [ false ]
Line 6: T_NOT_EQUALS [ != ]
Line 6: T_LPAREN [ ( ]
Line 6: T_ID [ a ]
Line 6: T_DOUBLE_EQUALS [ == ]
Line 6: T_ID [ a ]
Line 6: T_RPAREN [ ) ]
```

Verbose Mode: Off

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id a from line 2 into symbol table at scope: 0

Inserting id b from line 5 into symbol table at scope: 0

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Vervose Mode: ON

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

T_LBRACE expected!

T_LBRACE consumed!

T_INT consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_DIGIT expected!

T_DIGIT consumed!

T_BOOLEAN consumed!

T_ID expected!

T_ID consumed!

T_ID expected!

T_ID consumed!

T_SINGLE_EQUALS expected!

T_SINGLE_EQUALS consumed!

T_LPAREN, T_TRUE or T_FALSE expected!

T_LPAREN consumed!

T_LPAREN, T_TRUE or T_FALSE expected!

T_TRUE consumed!
```

CST

```
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Variable Declaration>
-----[int]
-----[a]
---<Statement List>
----<Statement>
-----<Assignment Statement>
------[a]
------[=]
------<Expression>
-------<Int Expression>
--------[1]
----<Statement List>
-----<Statement>
------<Variable Declaration>
-------[boolean]
-------[b]
-----<Statement List>
------<Statement>
-------<Assignment Statement>
--------[b]
--------[=]
--------<Expression>
---------<Boolean Expression>
----------[(]
----------<Expression>
-----------<Boolean Expression>
-----------[true]
----------[==]
----------<Expression>
-----------<Boolean Expression>
-----------[(]
------------<Expression>
-------------<Boolean Expression>
-------------[true]
------------[!=]
------------<Expression>
-------------<Boolean Expression>
-------------[(]
--------------<Expression>
---------------<Boolean Expression>
----------------[false]
--------------[==]
--------------<Expression>
---------------<Boolean Expression>
```

AST

```
<BLOCK>
-<Variable Declaration>
--[int]
--[a]
-<Assignment Statement>
--[a]
--[1]
-<Variable Declaration>
--[boolean]
--[b]
-<Assignment Statement>
--[b]
--<Equal>
---[true]
---<Not Equal>
----[true]
----<Equal>
-----[false]
-----<Not Equal>
------[true]
------<Not Equal>
-------[false]
-------<Equal>
--------[a]
--------[a]
-<Print Statement>
--[b]
```

Symbol Table

Name	Type	Scope	Line #
a	int	0	2
b	boolean	0	5

Test 6: Scope Test 1

Source Code

Select an example

Scope Test

Enter code

```
{
        int a
        a=9
        if (1 != 2) {
                int a
                a = 1
                if (a == 1){
                        int b
                        b=7
                        while (b != 7){
                                string c
                                c = "hello world"
                        }
                }
        }
} $
```

Verbose

Submit

This test is meant to show that the different levels of scope are recognized

AST

```
<BLOCK>
-<Variable Declaration>
--[int]
--[a]
-<Assignment Statement>
--[a]
--[9]
-<If Statement>
--<Not Equal>
---[1]
---[2]
--<BLOCK>
---<Variable Declaration>
----[int]
----[a]
---<Assignment Statement>
----[a]
----[1]
---<If Statement>
----<Equal>
-----[a]
-----[1]
----<BLOCK>
-----<Variable Declaration>
------[int]
------[b]
-----<Assignment Statement>
------[b]
------[7]
-----<While Statement>
------<Not Equal>
-------[b]
-------[7]
------<BLOCK>
-------<Variable Declaration>
--------[string]
--------[c]
-------<Assignment Statement>
--------[c]
--------[hello world]
```

Symbol Table

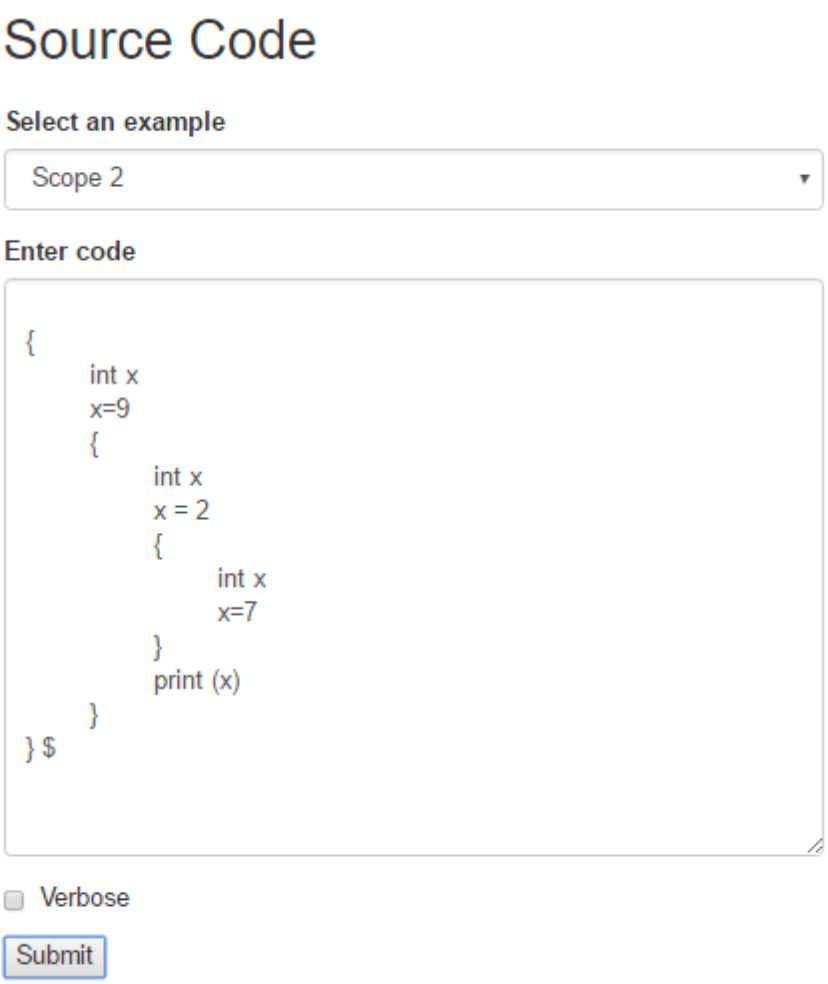
Name	Type	Scope	Line #
a	int	0	2
a	int	1	5
b	int	2	8
c	string	3	11

As seen on the previous page the compiler recognizes eash variable and places it in the symbol table accordingly along with the name of the variable, the type it is, which scope its in, and lastly what line number its on. If you look at the AST you can see the different levels of scope as they are based off of the indentation and how many dashes are displayed. The log also presents information based on the scope as well below.

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id a from line 2 into symbol table at scope: 0

Inserting id a from line 5 into symbol table at scope: 1

Inserting id b from line 8 into symbol table at scope: 2

Inserting id c from line 11 into symbol table at scope: 3

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Test 7: Scope Test 2



This test is meant to show that the different levels of scope are recognized

AST

```
<BLOCK>
-<Variable Declaration>
--[int]
--[x]
-<Assignment Statement>
--[x]
--[9]
-<BLOCK>
--<Variable Declaration>
---[int]
---[x]
--<Assignment Statement>
---[x]
---[2]
--<BLOCK>
---<Variable Declaration>
----[int]
----[x]
---<Assignment Statement>
----[x]
----[7]
-<Print Statement>
--[x]
```

Symbol Table

Name	Type	Scope	Line #
x	int	0	2
x	int	1	5
x	int	2	8

As seen on the previous page the compiler recognizes eash variable and places it in the symbol table accordingly along with the name of the variable, the type it is, which scope its in, and lastly what line number its on. If you look at the AST you can see the different levels of scope as they are based off of the indentation and how many dashes are displayed. The log also presents information based on the scope as well below.

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id x from line 2 into symbol table at scope: 0

Inserting id x from line 5 into symbol table at scope: 1

Inserting id x from line 8 into symbol table at scope: 2

Semantic Analysis Complete

Semantic Analysis produced 0 error(s) and 0 warning(s).
```

Test 8: Run All

Enter code

```
{
	int a
	a = 4

	int b
	b = 2 + a
} $

{
	int a
	a = 4
	if (a == 4) {
		print("hello world")
	}
} $

{
	if (1 == 1) {
		int a
		a = 1
	}
} $

{
	if (1 != 2) {
		int a
		a = 1
	}
} $

{
	int a
	a = 1

	if(a == 1) {
		a = 2
	}

	if(a != 1) {
		a = 3
	}
} $

{
	int x
	x = 0

	while (x != 5)
	{
		print(x)
		x = 1 + x
	}
} $
```

This test takes all the previous test and puts them into the source code as multiple programs to show that multiple programs can be run.

The output then as diplayed on the right is broken up into the different programs each individually with its own information displayed including Source Code, Tokens, CST, AST, Log, and Symbol Table

Output

Source code

```
{
        int a
        a = 4

        int b
        b = 2 + a
}$
```

Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_INT [ int ]

Line 2: T_ID [ a ]

Line 3: T_ID [ a ]

Line 3: T_SINGLE_EQUALS [ = ]

Line 3: T_DIGIT [ 4 ]

Line 5: T_INT [ int ]

Line 5: T_ID [ b ]

Line 6: T_ID [ b ]

Line 6: T_SINGLE_EQUALS [ = ]

Line 6: T_DIGIT [ 2 ]

Line 6: T_PLUS [ + ]

Line 6: T_ID [ a ]

Line 7: T_RBRACE [ } ]

Line 7: T_EOF [ $ ]
```

CST

```
<Program>
-<Block>
--[{]
--<Statement List>
---<Statement>
----<Variable Declaration>
```

Source code

```
{
        int a
        a = 4
        if (a == 4) {
                print("hello world")
        }
}$
```

Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_INT [ int ]

Line 2: T_ID [ a ]

Line 3: T_ID [ a ]

Line 3: T_SINGLE_EQUALS [ = ]

Line 3: T_DIGIT [ 4 ]

Line 4: T_IF [ if ]

Line 4: T_LPAREN [ ( ]

Line 4: T_ID [ a ]

Line 4: T_DOUBLE_EQUALS [ == ]

Line 4: T_DIGIT [ 4 ]

Line 4: T_RPAREN [ ) ]

Line 4: T_LBRACE [ { ]

Line 5: T_PRINT [ print ]

Line 5: T_LPAREN [ ( ]

Line 5: T_QUOTE [ " ]

Line 5: T_ID [ h ]

Line 5: T_ID [ e ]

Line 5: T_ID [ l ]

Line 5: T_ID [ l ]

Line 5: T_ID [ o ]

Line 5: T_WHITE_SPACE [   ]
```

Test 9: Type declaration Error

The compiler takes in the source code and creates the tokens however the parser realizes that a certain part in the source code doesn't agree with the grammer so it recognizes that and throws an error. Inside the error statement information regarding the error is displayed as well including the line number in which the error is located, the token the parser found, and lastly the token the parser was expecting. In this case the parser notices that after int the grammer is expecting a character between a-z as the identifier but instead it gets a digit so it throws an error

Output

Source code

```
{
        int 7
        a = 3
}$
```

Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_INT [ int ]

Line 2: T_DIGIT [ 7 ]

Line 3: T_ID [ a ]

Line 3: T_SINGLE_EQUALS [ = ]

Line 3: T_DIGIT [ 3 ]

Line 4: T_RBRACE [ } ]

Line 4: T_EOF [ $ ]
```

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Error on line 2: Found T_DIGIT, expected T_ID.
```

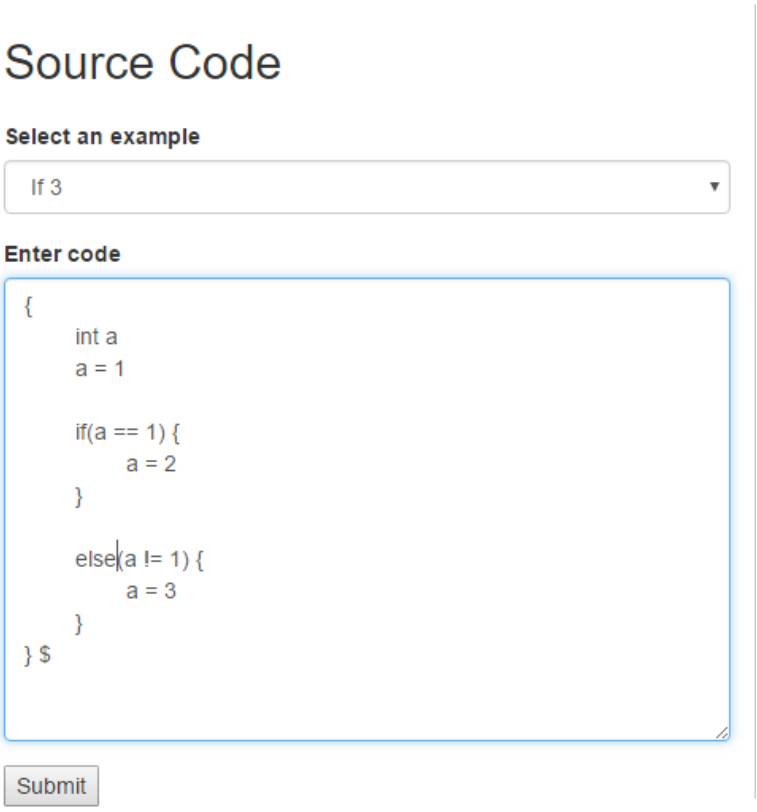
Test 10: Boolean Error

The compiler takes in the source code and creates the tokens however the parser realizes that a certain part in the source code doesn't agree with the grammer so it recognizes that and throws an error. Inside the error statement information regarding the error is displayed as well including the line number in which the error is located, the token the parser found, and lastly the token the parser was expecting. In this case the parser recogzines that only double equal (==) are allowed in boolean expressions so it throws an error.

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Error on line 4: T_SINGLE_EQUALS is not a valid boolean operator.
```

Test 11: Lexeme not in the Grammer E



The compiler takes in the source code and creates the tokens however the parser realizes that a certain part in the source code doesn't agree with the grammer so it recognizes that and throws an error. Inside the error statement information regarding the error is displayed as well including the line number in which the error is located, the token the parser found, and lastly the token the parser was expecting. In this example the parser recognizes that else is not a keyword in the grammer so it throws an error.

Output

Test program

```
{
        int a
        a = 1

        if(a == 1) {
                a = 2
        }

        else(a != 1) {
                a = 3
        }
} $
```

Error

```
Lexical Error on line 9: else is not a valid lexeme.
```

Test 12: Missing Brace/Parenthesis Error

The compiler takes in the source code and creates the tokens however the parser realizes that a certain part in the source code doesn't agree with the grammer so it recognizes that and throws an error. Inside the error statement information regarding the error is displayed as well including the line number in which the error is located, the token the parser found, and lastly the token the parser was expecting. In this example the Right brace at the bottom of the code is missing so the parser through an error and placed the line number to where the right brace would be expected.

Source code

```
{
        int a
        a = 4

        int b
        b = 2 + a$
```

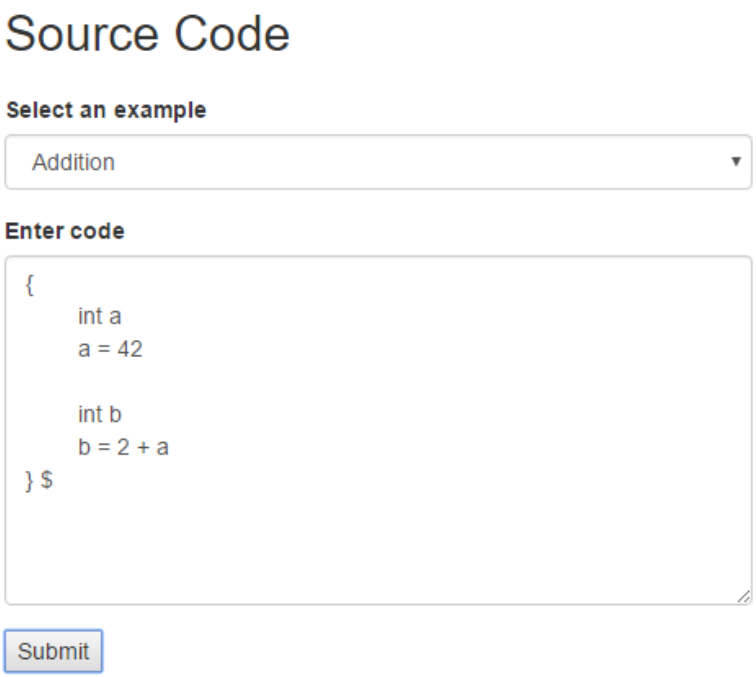
Tokens

```
Line 1: T_LBRACE [ { ]

Line 2: T_INT [ int ]

Line 2: T_ID [ a ]

Line 3: T_ID [ a ]

Line 3: T_SINGLE_EQUALS [ = ]

Line 3: T_DIGIT [ 4 ]

Line 5: T_INT [ int ]

Line 5: T_ID [ b ]

Line 6: T_ID [ b ]

Line 6: T_SINGLE_EQUALS [ = ]

Line 6: T_DIGIT [ 2 ]

Line 6: T_PLUS [ + ]

Line 6: T_ID [ a ]

Line 6: T_EOF [ $ ]
```

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Error on line 6: Found T_EOF, expected T_RBRACE.
```

Test 13: Integer over digit Error



The compiler takes in the source code and creates the tokens however the parser realizes that a certain part in the source code doesn't agree with the grammer so it recognizes that and throws an error. Inside the error statement information regarding the error is displayed as well including the line number in which the error is located, the token the parser found, and lastly the token the parser was expecting. In this example a type int is declared and the value 42 is to be assinged to a however only single digits can be assigned to identifiers so the parser recogizes that and throws an error.

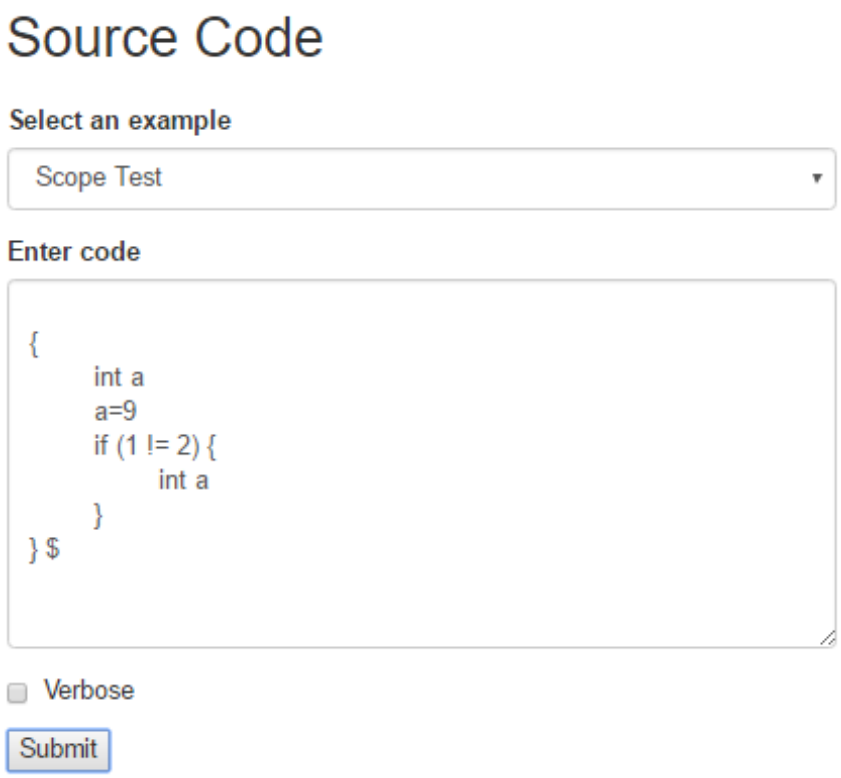
Test program

```
{
        int a
        a = 42

        int b
        b = 2 + a
} $
```

Error

```
Lexical Error on line 3: 42 is not a valid lexeme.
```

Test 14: Non-initialized variables (Scope Warning)



In the above example we are wanting to see how the compiler will handle an variable that is declared but not initialized with a value. Here the compiler will continue as normal but in the log there will be warning in green that appear indicating that

Log

```
Performing Lexical Analysis

Lexical analysis produced 0 error(s) and 0 warning(s)

Performing Parsing

Parsing Complete

Parsing produced 0 errors and 0 warnings

Performing Semantic Analysis

Inserting id a from line 2 into symbol table at scope: 0

Inserting id a from line 5 into symbol table at scope: 1

Semantic Analysis Complete

Warnings

Warning! The id a declared on line 5 was declared, but never used.
Warning! The id a declared on line 5 was never initialized.
Semantic Analysis produced 0 error(s) and 2 warning(s).
```

As seen above in the text in green. Warning as issued indicating that the id declared on a specific line was never used however it did not interfere with the compilers other function which is why it remains a function.