2010

Simple Interpreted Language

Final Documentation

Authors: Peter Cappello
Damon Chastain
Matthew Urtnowski

Course: CECS 543
Instructor: Dr. Michael Hoffman
Date: Fall 2010

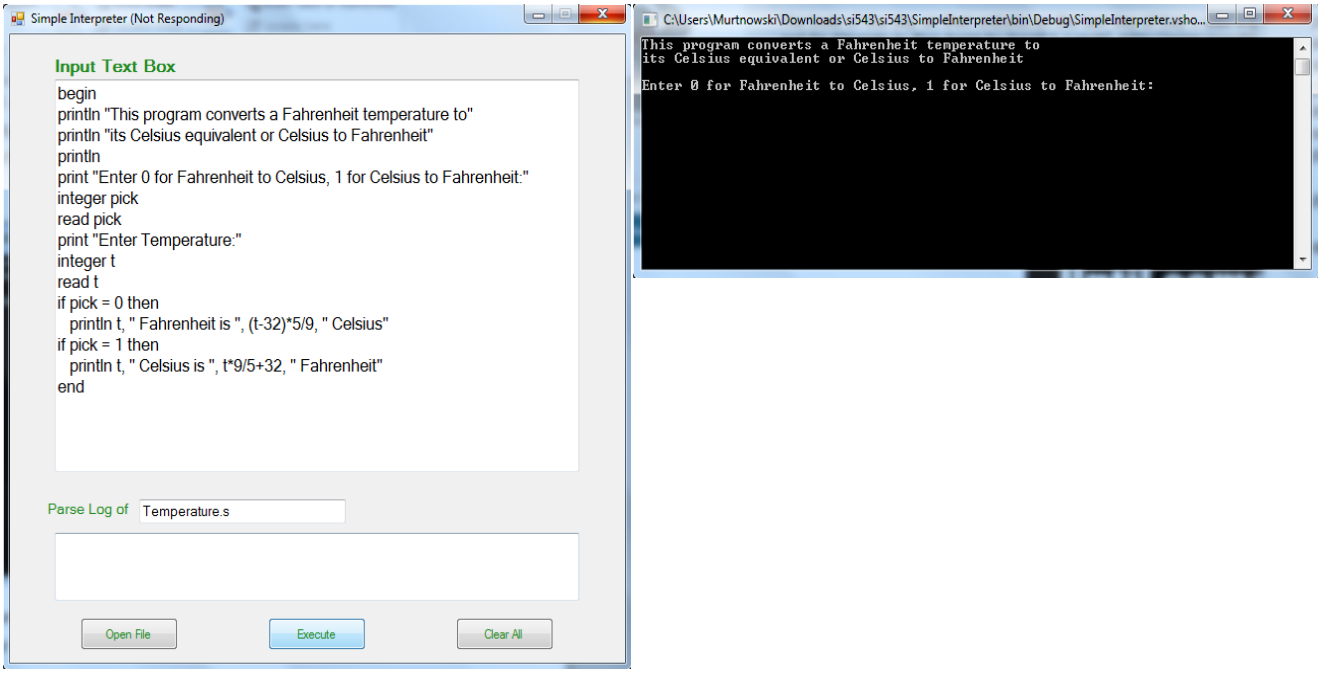
Table of Contents

1. Description

Simple Interpreted Language or SIL is a simple language created for a California State University, Long Beach CECS 543 course. It is accompanied by a rudimentary development tool than runs in a Windows GUI. The Windows GUI can be used to load SIL code stored in files with a .S extension or code can be edited directly in the GUI. The Windows GUI also allows SIL code to be executed in a Console. All SIL Programs display output to the Console and collect input from the Console. The SIL project is capable of detecting and managing compile-time and run-time errors while running SIL Programs.

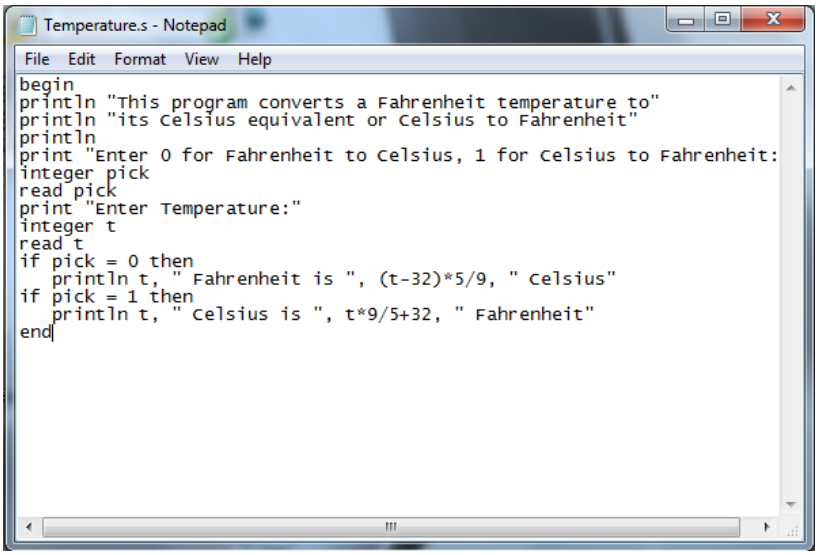
2. Interface

2.1 Description



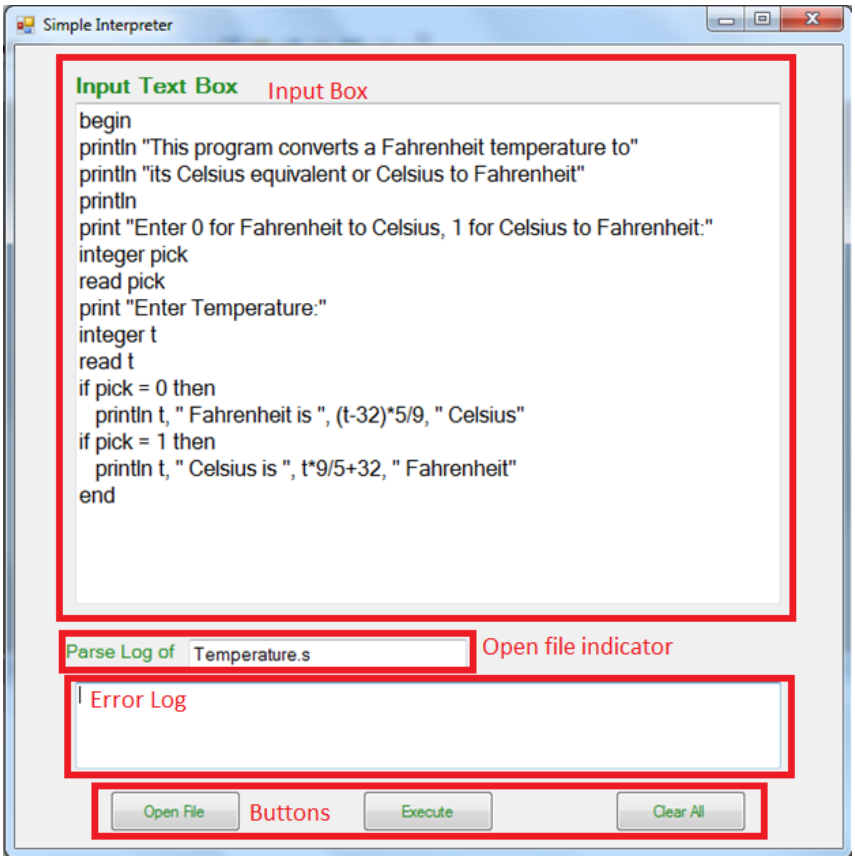
Users may interface with our SIL project in three methods. First is the creation of .S files that contain SIL plain text code ready to be executed. Second is using a Windows GUI to load .S files or type in SIL code directly. The Windows GUI is also used to execute code and display any syntax or runtime errors. The third way is by a simple Console, where the actual SIL program displays output and collects input.

2.2 Input File



Simple Interpreted Language code may be stored in a plain text file with a .S extension. .S files are made up of common ASCII characters and can be open with any plain text editors such as Notepad.

2.3 Window GUI



The Windows GUI is a form of rudimentary Interactive Development Environment for our Simple Interpreted Language. The user can use it to load .S code files, write code, initiate the running of a SIL program, and read any error messages that occur.

2.3.1 Input Box

The Input Box is an editable textbox that uses input code for executing our Simple Interpreted Language. If the user chooses to load a .S file instead of typing their code manually, the contents of that file will be shown in the Input Box. The code will still be editable after loading a .S, but any changes in the code are temporary and will not change the original .S file.

2.3.2 Opened File Indicator

This is a textbox that is used to display the title of the most recently loaded .S file.

2.3.3 Error Log

Any syntax or runtime errors are displayed in this text box. For a list of supported SIL errors and their associated messages see Section 3.7.

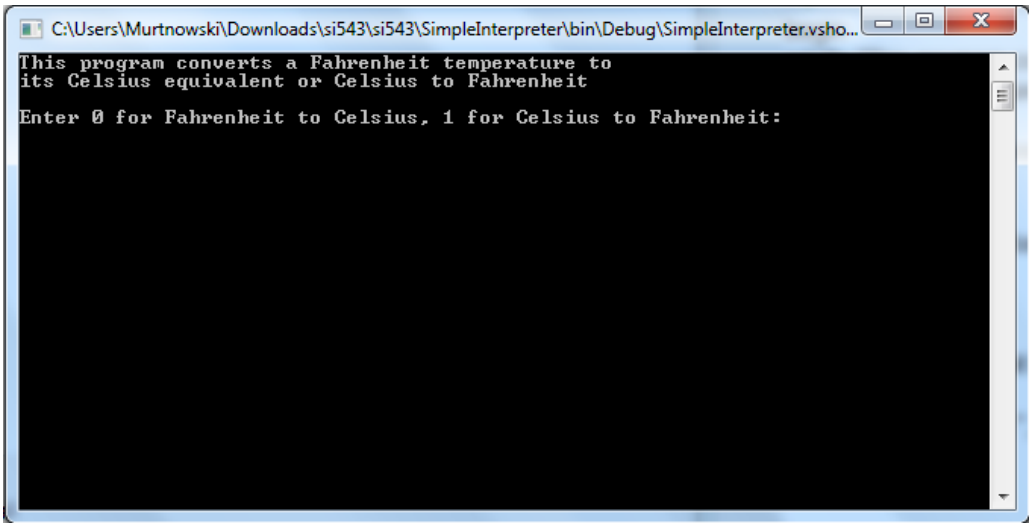
2.3.4 Buttons

There are three buttons on the Window GUI that the user may invoke.

The first button is 'Open File' that opens a new openfile window where a user may navigate the local file system and select a .S file. The openfile window is restricted to only show files with the .S extension. If the user selects a file to open, the contents of that file will be used to populate the Input Box (Section 2.3.1).

The second button is the 'Execute' button, which causes the code in the Input Box (Section 2.3.1) to be executed and the SIL program to be run in the Console Window (Section 2.4). If there is a syntax error, the code will not execute and any errors will be displayed in the Error Log (Section 2.3.3).

2.4 Console



An executed SIL program is displayed in the Console Window. All user input is collected by this user, and any output displayed by the program is displayed here.

3. SIL Characteristics

3.1 Symbols

3.1.1 Constant

An integer value constant with a range of a signed 32 bit number. An example might be the number 42 or the number -314.

3.1.2 Literal

A string of ASCII characters no greater than ASCII 126 is encapsulated by double quotations. The length of the literal may not exceed 80 characters including the two double quotations. An example of a literal might be “The quick brown fox ran fast.” or an empty literal such as “”.

3.1.3 Operator

Operators are special characters that serve a special purpose

Operator:	Purpose:
=	Equal assignment or relational equality
>	Greater than comparison
<	Less than comparison
(	Open parenthesis for determining arithmetic order of operation
)	Close parenthesis for determining arithmetic order of operation
+	Arithmetic addition
-	Arithmetic subtraction
*	Arithmetic multiplication
/	Arithmetic division
%	Modulus Operator
,	Comma list separator
[	Open bracket for initializing an array
]	Close bracket for initializing an array
>=	Greater than equal
<=	Less than equal
!=	Not equal
AND	Logical and
OR	Logical or
#	Comment indicator

3.1.4 Identifier

An Identifier is a user defined string to logically identify a variable name. All Identifiers are of a length inclusively between 1 and 32 characters. Any Identifier used in a SIL program is stored in a runtime Identifier Table (Section 3.3) along with its current value.

3.1.5 Reserved Words

A list of reserved words is defined to denote the start of a program statement. These words are: INTEGER, LET, PRINT, PRINTLN, READ, IF, THEN, WHILE, FOR, BEGIN, END, DECLARE, ADD, and FUNCTION.

3.2 Data Types

3.2.1 Integer

An integer is a 32 bit signed number.

3.2.2 String

A string is a set of ASCII characters that may not exceed 80 characters. Only ASCII characters greater than ASCII 126 may be included in a string.

3.2.3 Boolean

A data type that may either have a logical value of TRUE or FALSE.

3.2.4 Double

A double number with a 32 bit signed range.

3.2.5 Array

An array is a collection of other non-array data types defined in section 3.2. An array is initially declared with a size of zero then grows as elements are added to it. An array may grow to a size of a signed 32 bit integer.

3.3 Identifier Table

All variables used in the SIL program will have their Identifiers and associated values stored in a table created during runtime. This table supports up to 8,192 variables. The relation between the Identifier and value is stored with fast lookup as the

primary concern. Four types of interactions can be made by the Identifier table, which are as follows: Does an Identifier exist in table, Insert new Identifier into the table, Return a value of Identifier, and Set value of Identifier.

3.3.1 Does an Identifier exist in table

Query where the Identifier should be in the table. If the Identifier exists in the table, a logical TRUE is returned. If the Identifier is not found in the table, a logical FALSE is returned.

3.3.2 Insert new Identifier into the table

Query where the Identifier should be in the table and insert it into the table. After the Identifier has been inserted in the table, return a logical value of TRUE. If the Identifier already exists in the table, return a 'Variable already declared' error as defined in Section 3.7.2. If inserting the Identifier causes the number of stored variables to exceed a total of 8,192 variables, the variable is not inserted into the table and a 'Maximum stored variables' error as, defined in Section 3.7.2, is returned.

3.3.3 Return value of Identifier

Query where the Identifier should be in the table and, if the Identifier exists in the table, return the value of the Identifier. If the Identifier is not found a 'Variable not declared' message is returned as defined in Section 3.7.2.

3.3.4 Set value of Identifier

A 'Does an Identifier exist in table' (Section 3.3.1) query is completed first. If the Identifier exists in the table, a new value is assigned for that Identifier. If that Identifier does not exist in the table, an 'Undeclared Identifier' (Section 3.7.2) error is returned.

3.4 SIL Program Statements

All SIL Programs begins with a Begin … End Statement (Section 3.4.9)

3.4.1 Integer Statement

INTEGER *identifier [, identifier]*

The Integer Statement adds a new integer variable to the Identifier Table (Section 3.3). More than one integer identifier may be initialized at once by adding commas between the Identifiers. If the Identifier already exists in the Identifier Table (Section 3.3) then a 'Variable already declared' (Section 3.7.2) occurs.

3.4.2 Let Statement

LET *identifier = expression*

The Let Statement updates the value of an Identifier already in the Identifier Table (Section 3.3). The value assigned to the Identifier is the assessment of the expression (Section 3.5). If the Identifier does not exist, a 'Variable not declared' (Section 3.7.2) occurs.

3.4.3 Print Statement

PRINT [[*expression* | *literal*] [,*expression* | *literal*]]

The Print Statement is used to output the assessment of an expression or a string literal to the console. The expression is defined in Section 3.5 and literal is defined in

Section 3.1.2. The output is written onto a line of the Console and it does not extend to the next line upon completion. The Print Statement may output multiple expressions or literals by adding commas between the expressions/literals.

3.4.4 Println Statement

PRINTLN [[*expression* | *literal*] [,*expression* | *literal*]]

The Println Statement is exactly like the Print Statement (Section 3.4.3), except at the completion of the statement of the output, future statements added to the console areexpressed on the next line in the console. An empty Println Statement can be used to create a line break in the console.

3.4.5 Read Statement

READ *identifier*

The Read Statement is used to collect input from the user via the keyboard and the result is stored into the Identifier. After the user types his input, the user then clicks enter to signify completion. The input must be a valid value for the type of the Identifier. If the value does not match an accepted value for the variable then a 'Not in a correct format' error occurs as defined in Section 3.7.2.

3.4.6 If … Then Statement

IF *relational_expression* THEN

statement

The If … Then Statement is used to execute a statement if relational_expression is assessed TRUE. If the relational_expression is assessed to be FALSE, the statement does

not execute. Relational_expression is defined in Section 3.6. Statement is defined in Section 3.4.

3.4.7 While Statement

WHILE *relational_expression*

statement

ENDWHILE

The While statement is used to repeatedly execute a Statement (Section 3.4) while the relational_expression (Section 3.6) is evaluated to be TRUE. The relational_expression is first evaluated. If the relational_expression is TRUE, the statement is executed. The relational_expression is then reevaluated to determine if the statement should be executed again. If the relational_expression is evaluated to be false, the statement is not executed.

3.4.8 For Statement

FOR *identifier* = *constant* TO *identifier*

Statement

NEXT

The For Statement is used to execute a statement for a particular number of times. The Identifier (Section 3.1.4) before the 'TO' in the statement is initialized by a Constant (Section 3.1.1) and then increments each time the NEXT is reached after the Statement (Section 3.4) is executed. The first identifier is compared against the second identifier such that it evaluates TRUE when the first is less than or equal to the second.

An evaluation of TRUE will cause the statement to execute. An evaluation of FALSE will cause the Statement to not be executed and the program will execute lines after the For Statement. The identifier after the 'TO' in the statement must be an identifier associated with an integer.

3.4.9 Begin … End Statement

BEGIN

Statement1

Statement2

…

StatementN

END

The Begin Statement is used to define a block of Statements.

3.4.10 Function Statement

FUNCTION *Identifier* (*integer parameter list*)

RETURN *Statement*(s)

The Function Statement is used to define a function that can be called which returns an integer. The function can support an unlimited number of integer parameters. Functions can only be invoked by expressions and cannot be used as a standalone statement.

3.4.11 Declare Array Statement

DECLARE*identifier*[0]

The Declare Statement is used to create a new Array. An array can be used to store a collection of data types defined in Section 3.2.

3.4.12 Add to Array Statement

ADD(*identifier*, *constant*|*literal*)

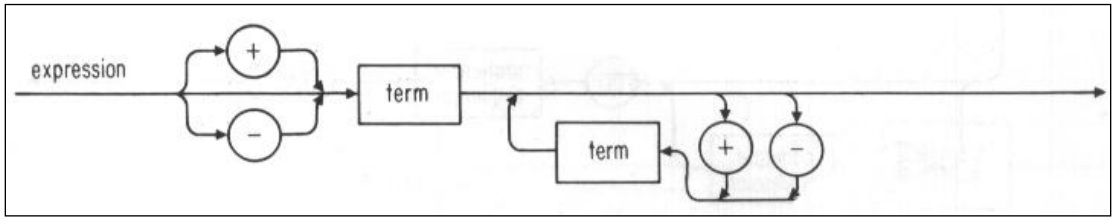
The Add statement is used to add append a new constant or literal to the end of an array. The array to be manipulated is defined by the identifier. The data to add to the array is after the second parameter of the Add Statement. The array must be declared before the Add Statement can be used on it.

3.4.13 Array Length Statement

LENGTH (*identifier*)

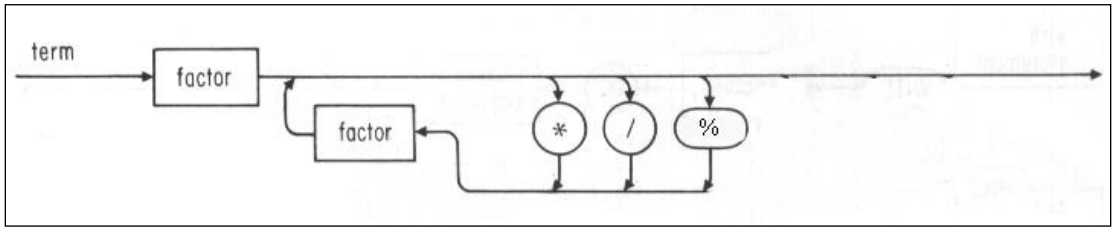
The Array Length Statement is used to return the length of a particular Array identified by the identifier.

3.5 SIL Expression



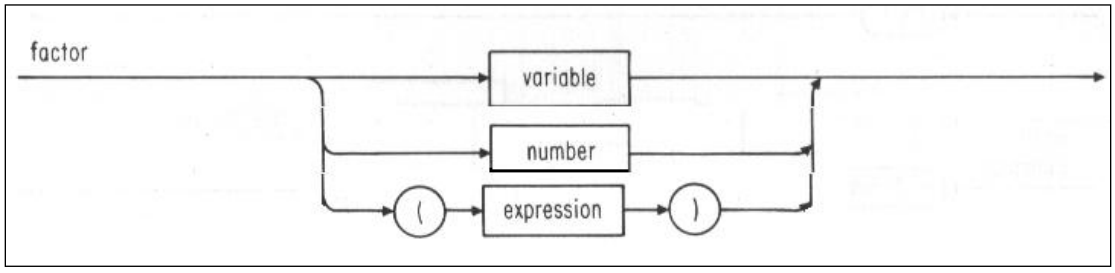
The diagram above denotes a flow chart for how an expression can be represented.

3.5.1 Term



The diagram above denotes a flow chart for how a Term can be represented.

3.5.2 Factor



The diagram above denotes a flow chart for how a Factor can be represented.

3.6 Relational Expression

Expression { = | < | > | <= | >= | != } *Expression*

A relational expression is a comparison of the values of two expressions (Section 3.5). The result of a relational expression is either TRUE or FALSE. The expressions must have a numeric value.

3.7 SIL Errors

3.7.1 Syntax Errors

Any Syntax Errors are caught before the program executes and causes an error message displaying the line number and type of error in the Error Log (Section 2.3.3).

Type	Occurrence
Missing THEN statement	An IF statement without a THEN clause
Type is undefined	Declaring a variable type that is not defined
Identifier may not exceed 32 Characters in length	Creating an identifier that is long that 32 characters
Undeclared identifier	Trying to use an identifier that has not been declared
Variable already declared	Trying to declare a variable that already exists in the identifier table
String may not exceed 80 characters	Creating a string longer than 80 characters
Missing BEGIN statement	Starting a SIL program without a Begin Statement
Generic Error	Other errors that are trapped show a generic error message

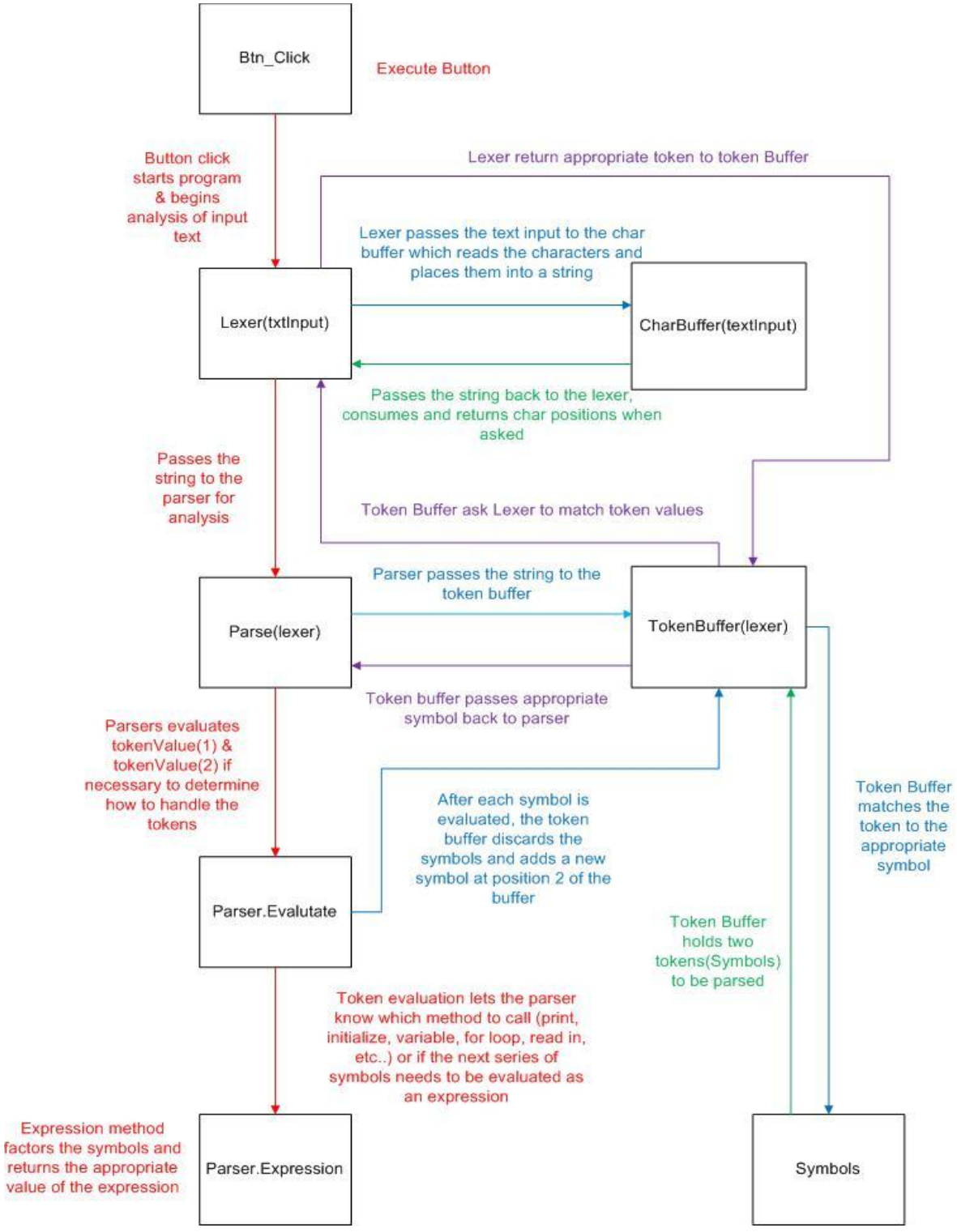
3.7.2 Runtime Errors

Any runtime errors will cause the SIL Program to stop running and display an error message displaying the line number and type of error in the Error Log (Section 2.3.3).

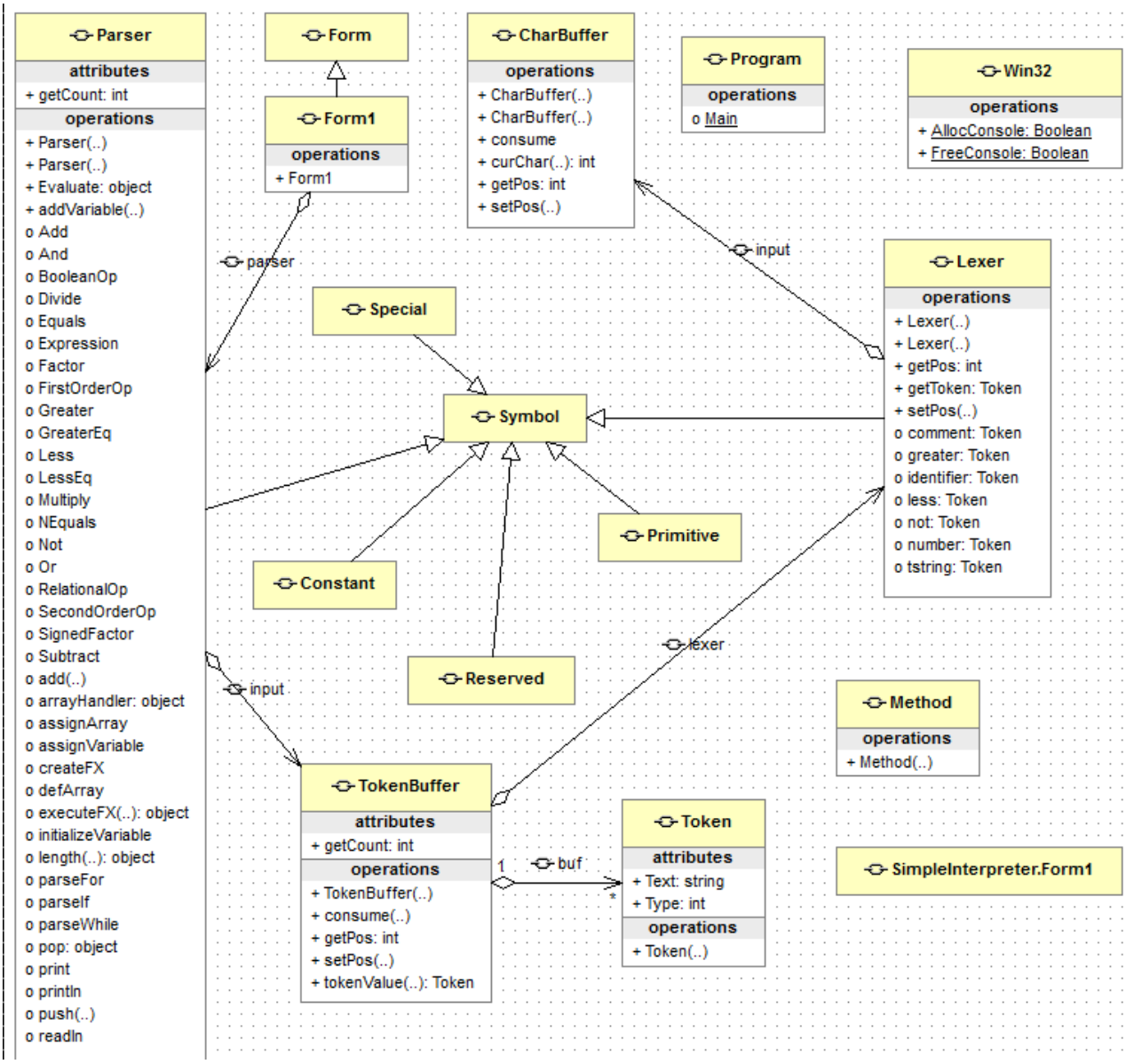
Type	Occurrence
Divide by zero	When a number is divided by zero
Identifier Table Full	When the identifier table exceeds 8,192
Numeric Overflow	When a number outside of the signed 32 bit range is stored

4. Interpreter

The actions taken by our interpreter are fairly complex. There are three main components of our code: A GUI, A Lexer, and a Parser. The easiest way to show the interaction between these components is a graphic diagram below.



A class diagram of our code is seen in the graph below



5. Bonus Features

Several bonus features to the SIL requirements have been implemented. We have listed them throughout this document, but we created this section so we could specifically enumerate them. A list of the complete features is below.

1. Multiple identifiers can be declared with one statement.
2. Multiple expressions and literals can be printed with one statement.
3. Comparison operations >= , <= , and != be implemented in boolean statements.
4. Boolean operators AND and OR can be used in boolean comparisons.
5. Additional data types Boolean, String, Double and Array can be used.
6. SIL uses step by step execution in parsing.
7. Uses '#' to indicate a line comment

6. Sample Code

6.1 Temperature

6.1.1 SIL Code

```
begin

println "This program converts a Fahrenheit temperature to"

println "its Celsius equivalent or Celsius to Fahrenheit"

println

print "Enter 0 for Fahrenheit to Celsius, 1 for Celsius to Fahrenheit:"

integer pick

read pick

print "Enter Temperature:"

integer t

read t

if pick = 0 then

  println t, " Fahrenheit is ", (t-32)*5/9, " Celsius"

if pick = 1 then

  println t, " Celsius is ", t*9/5+32, " Fahrenheit"
```

end

6.1.2 Ouput

6.1.2.1 Error Log

None there were no errors

6.1.2.2 Console Output

```
This program converts a Fahrenheit temperature to
its Celsius equivalent or Celsius to Fahrenheit

Enter 0 for Fahrenheit to Celsius, 1 for Celsius to Fahrenheit:1
```

```
Enter Temperature:100
```

```
100 Celsius is 212 Fahrenheit

Successful run
```

6.2 Circle

6.2.1 SIL Code

```
begin

println "This program calculates the area and circumference"

println "of a circle to an integer approximation"

println

print "Enter Radius:"

integer radius, area, circum

read radius

let circum = (radius + radius) * 22 / 7

let area = radius * radius * 22 / 7

println

println "Area = ", area

println "Circumferenece = ", circum

end
```

6.2.2 Output

6.2.2.1 Error Log

None there were no errors.

6.2.2.2 Console Output

```
This program calculates the area and circumference
of a circle to an integer approximation

Enter Radius:5
```

```
Area = 78
Circumferenece = 31

Successful run
```

6.3 Wage

6.3.1 SIL Code

```
begin

println "This program calculates hourly salary with overtime"

println "for any full hour over 40. Overtime is 1.5"

println

integer rate, hours, pay

print "Enter number of hours:"

read hours

print "Enter rate of pay:"

read rate

if hours > 40 then

  let pay = 40 * rate + ((hours - 40) * rate * 3 / 2)

if hours = 40 then

  let pay = 40 * rate

if hours < 40 then

  let pay = hours * rate

println
```

```
println "Pay for ", hours, " at $", rate, " per hour is $", pay

end
```

6.3.2 Output

6.3.2.1 Error Log

None there were no errors

6.3.2.2 Console Output

```
This program calculates hourly salary with overtime
for any full hour over 40. Overtime is 1.5

Enter number of hours:40
```

```
Enter rate of pay:20
```

```
Pay for 40 at $20 per hour is $800

Successful run
```

6.4 Array Test

6.4.1 SIL Code

```
begin

declare gg[0]

add(gg, "hi")

add(gg, "this")

add(gg, "works")

add(gg, "maybe")

integer jj, kk

let jj = length(gg)

for kk = 0 to jj

println gg[kk-1]

next

end
```

6.4.2 Output

6.4.2.1 Error Log

None there were no errors

6.4.2.2 Console Output

```
this
hi
this
works
maybe

Successful run
```

6.5 Simple Function

6.5.1 SIL Code

```
begin

integer radius, area

let radius = 7

function Area(r)

 return r * r * 22 / 7

let area = Area(radius)

print area

end
```

6.5.2 Output

6.5.2.1 Error Log

None there were no errors

6.5.2.2 Console Output

```
154
Successful run
```

6.6 Nested Loops

6.6.1 SIL Code

```
begin

integer x, z
```

```
for x = 0 to 2

print "x loop"

for z = 0 to 5

println z

next

next

end
```

6.6.2 Output

6.6.2.1 Error Log

None there were no errors.

6.6.2.2 Console Output

6.7 Nested Conditional

6.7.1 SIL Code

```
begin

integer x, z

let x = 5

let z = 0

if x > 3 then
```

```
if z < 2 then

if z < 1 then

print x

end
```

6.7.2 Output

6.7.2.1 Error Log

None there were no errors.

6.7.2.2 Console Output

```
5
Successful run
```