Compiler for a Fortran-Like Programming Language

User Manual

Prepared for

Dr. Chen-Wei Jackie Wang

EECS4302 - Compilers and Interpreters

By

Shivam Patel - 216385247
shivam08

Colin D'Souza - 216283582
colindis

William Closson - 216344285
closws

December 6, 2022

1. Input Language

1.1 Structure of a Program

```
PROGRAM fibonacci

	FUNC fib(INT :: n) :: INT
		IF (n <= 2) THEN
			n = 1
		END IF

		IF (n > 2) THEN
			n = fib(n - 1) + fib(n - 2)
		END IF

		RETURN n
	END FUNC

	!! entry point of program execution
	FUNC main()
		INT :: a
		BOOL :: b
		BOOL :: c
		c = TRUE
		IF ((!b && c) && getTrue()) THEN
			IF (a == 0 || FALSE) THEN
				printFib(6)
			END IF
		END IF
	END FUNC

	FUNC printFib(INT :: n)
		PRINT(fib(n))
	END FUNC

	FUNC getTrue() :: BOOL
		RETURN TRUE
	END FUNC

END PROGRAM
```

Figure 1: Structure of Input Program

1.1.1 Program Block

```
PROGRAM basic_program_template01
      !! Section: functions before main()
      FUNC main()
            !! all programs must have a main function
            !! it is entry point of program execution
      END FUNC
      !! Section: functions after main()
END PROGRAM
```

Figure 2: Most Basic Syntactically-Correct Program

In order to create an input program, developers must start with declaring the program block which starts with the keyword PROGRAM followed by the name of the program consisting of lowercase or uppercase letters, numbers, and or underscores. All block constructs must have an ending, which can be signified using the keyword END followed by the type of construct. In the case of the program block, the end is denoted using the keywords END PROGRAM.

Section: Functions

Functions can be declared in any order, however, all programs must have the main() function which is the entry point of execution. The name of the main function is case-sensitive, which means "mAiN()" is semantically incorrect, and it should not have any parameters or a return type.

1.1.2 Function Block

```
!! function signature
FUNC add(INT :: x, INT :: y) :: INT
      !! Section: variable declarations
      INT :: sum

      !! Section: instructions
      sum = x + y

      !! Section: return statement
      RETURN sum
END FUNC
```

Figure 3: Function Block

In order to create a function within a program, developers must start with declaring the function block which starts with the keyword FUNC followed by the name of the function which must follow the same constraints as the program name. Following the function name, the developer needs to specify 0 or more input parameters in a pair of parentheses and lastly, the optional return type. The end of the function block is denoted using the keywords END PROGRAM.

```
FUNC <name>(<parameters>) <return type>
```

Figure 4: Function Signature

Section: Variable Declarations

The first section within the function block is variable declarations which follow the format TYPE :: ID, where TYPE is the type of the variable which can be INT or BOOL and ID is the name of the variable which follows the same constraints as the program name. Function parameters are essentially variable declarations since they follow the same format as well. Not all functions need to declare additional variables, so this section is optional.

Section: Instructions

After the variable declaration section, there is a section for instructions which can include statements such as variable assignments, conditional blocks, print, function calls, etc… Most of the function's logic resides within this section.

Section: Return Statement

The last section is for methods with a return type. All methods with a return type must include a return statement which starts with the keyword RETURN followed by an expression which evaluates to a value that is the same type as the function's return type. Functions without a return type would not have this section.

1.2 List of Advanced Programming Features

1.2.1 Functions

Syntax of a function call is similar to any other programming language. Function call starts with the name of the function that you are calling, followed by a list of arguments enclosed in parentheses. The types of the arguments must match with the type of the parameters listed in the function's signature. At runtime, the values of the arguments will be passed over to the function and the rest of the function's instructions will be executed.

Function calls are unique in the sense that they are treated like expressions. In other words, functions with a return type can be used in conditionals, variable assignments, return statements, and as a plain old instruction as well. In order for a function call to be semantically correct, its return type must match its use case. For example, if you are using a function in variable assignment the return type of the function must be the same as the return type of the variable.

Examples of function calls can be found in example_3_functions.txt and example_4_function_returns.txt

1.2.2 Recursion

```
PROGRAM fibonacci

	FUNC fib(INT :: n) :: INT
		IF (n <= 2) THEN
			n = 1
		END IF

		IF (n > 2) THEN
			n = fib(n - 1) + fib(n - 2)
		END IF

		RETURN n
	END FUNC

	FUNC main()
		PRINT(fib(6))
	END FUNC
END PROGRAM
```

Figure 2: Most Basic Syntactically-Correct Program

Syntax of a function has been explained in Section 1.1.2 since it is the critical part of the structure of a program. In the example above, we use an integer function to recursively compute the n-th fibonacci number. Examples of recursion can be found in example_8_fibonacci.txt and example_6_division_modulo.txt. Examples of functions can be found in almost all example program files.

1.3 Structure of a Test

```
TEST example

    @Test
    FUNC test_1()

       main()

       ASSERT(getTrue())

    END FUNC

    @Test
    FUNC test_2()

       ASSERTEQUALS(fib(1), 1)
       ASSERTEQUALS(fib(6), 8)

    END FUNC

END TEST
```

In order to create a test program, developers must start with declaring the test block which starts with the keyword TEST followed by the name of the program consisting of lowercase or uppercase letters, numbers, and or underscores. All block constructs must have an ending, which can be signified using the keyword END followed by the type of construct. In the case of the program block, the end is denoted using the keywords END TEST. Within the test block, test functions can be declared in any order, however, all functions must have a @Test annotation. Syntax of the function call is similar to the programming language.

Within each test function block, you can call functions from the program file or use built in assertions which help check the validity of function calls with return value. The ASSERT function is used to check if the value is true. ASSERT will fail if the value provided is not true, making the entire test case/function fail as well. The ASSERTEQUALS function is used to compare two different values. ASSERTEQUALS will fail if the values provided are not equal. The ASSERT function takes 1 argument of type BOOL and ASSERTEQUALS takes two arguments of the same type either BOOL or INT.

2. Output Structure

The output is an HTML document that contains the following tabs. Each tab has the source code of the program, the test program, and sometimes an extra table with more in-depth details. For more information on how each coverage criteria works, see **Section 3**.

1. **Assertions**

This tab displays the results of the actual test program. The assertions in the source code of the test program are highlighted based on whether they passed or not.

2. **Statements**

This tab displays the results of “Statement Coverage”. Each line in the program source code is highlighted based on if it was run or not.

3. **Decisions**

This tab displays the results of “Decision Coverage”. Each decision (condition of an if-statement) in the program source code is highlighted based on if it was evaluated as true, false, both, or neither.

4. **All Defs**

This tab displays the results of “All-Defs Coverage”. Each declaration and function parameter is highlighted based on if it was covered by this criteria. A table gives a detailed list of each variable and checks off which once were covered.

5. **All C-Uses**

This tab displays the results of “All C-Uses Coverage”. Each C-Use in the program is highlighted based on if it was covered or not. A table lists all C-Uses in full detail.

6. **All P-Uses**

This tab displays the results of “All P-Uses Coverage”. Each P-Use in the program is highlighted based on if it was covered or not. A P-Use can be covered as true or false, and are highlighted in different colours accordingly. The table lists all P-Uses in full detail.

3. Justification of Output

What follows are more details for each code coverage criterion, explaining how they work. The output should correspond to these descriptions, confirming the accuracy of our compiler.

1. **Assertions**

A green highlight means the assertion passed, and red means it failed. This one is fairly straightforward.

2. **Statements**

For each statement in the program, a green highlight means the line was run at least once, and red means it was never run at all. This is clearly shown in the output.

3. **Decisions**

For each decision (condition of an if-statement), a green highlight means that it was evaluated as both TRUE and FALSE at some point during the run-time of the program. A yellow highlight means that it was only evaluated as either TRUE or FALSE, but not both. A red highlight means it was never run once.

4. **All Defs**

This coverage criteria looks at definition-clear paths. Each definition-clear path (DC Path) starts at a definition (e.g. declaration, assignment, etc…) and ends at a C-Use or P-Use. We will cover what those mean in their respective sections. But, each DC path involves a variable. So, we highlight the location where the variable was declared (either in a function or in its signature). When a variable is highlighted green, that means at least one of its DC Paths was covered: meaning, that both the definition and C-Use/P-Use was run. When a variable is highlighted red, none of its DC Paths are covered.

5. **All C-Uses**

This coverage criteria looks at DC Paths (defined in All Defs above) that end with a C-Use (or computation use). This includes being used on the right side of an assignment, a print statement, a return statement, or otherwise. So, to display which C-Uses have been covered, we highlight the line containing the actual C-Use itself. It's highlighted GREEN if it was covered, and RED if it was not.

6. **All P-Uses**

This coverage criteria looks at DC Paths that end with a P-Use (or predicate use). These are just when a variable is used in the condition of an if-statement. So, to display which P-Uses have been covered, we highlight the line containing the P-Use itself. Keep in mind that a P-Use can be evaluated as either TRUE or FALSE. We highlight it GREEN when both TRUE and FALSE are covered, YELLOW if it's only one, and RED if it's not covered at all.

4. Summary of Submitted Examples

We have included 5 examples that are proofs of concept for basic programming features, and the 5 additional examples that are more complex and solve real programming problems.

Basic Examples

4.1 Assignment & Printing

- Demonstrates that variable assignment and the print function works.
- Print function will print to the standard output (terminal).

4.2 Conditionals

- Demonstrates that if-statements work, including when they are nested.
- It's also an example of when All-Defs is not fully covered. (INT :: a)

4.3 Function Calls

- Demonstrates that basic function calls work.

4.4 Function Usage

- Demonstrates all the ways that function calls can be incorporated into other statements. Including: expression, print statements, if-statements, etc…

4.5 Assertions

- Demonstrates assertions and how they are used in the test language.

Application Examples

4.6 Integer Division and Modulus

- Because our programming language only supports integers, we chose not to add a native division or modulus operator. However, using recursion and subtraction, they can still be implemented in our programming language. This is what is demonstrated in this example.

4.7 Max of Three Numbers

- This is meant to show the use of deeply nested if-statements. It acts as a fairly complex example where our "All P-Uses" and "Decision Coverage" criteria are really put to the test.
- Another interesting feature of this example is that: even though we test it on all possible combinations of inputs, some parts of the function are never reached. Thus, our test language has literally found dead code in the function.

4.8 Fibonacci Sequence

- A classic example of a recursive function. It has two recursive calls in the same line, thus putting our call stack to the test.

4.9 GCD and LCM

- A recursive implementation of GCD and a clever LCM function that depends on the GCD function. Both of those functions depend on the previously defined division of modulo functions.

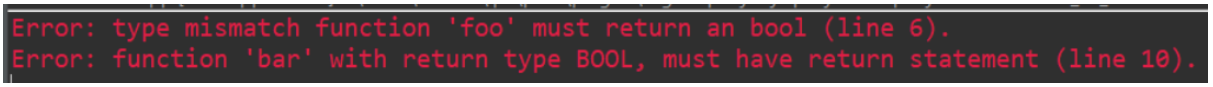
4.10 Complex Test Cases

5. Miscellaneous Features

5.1.1 Semantic Error Checking

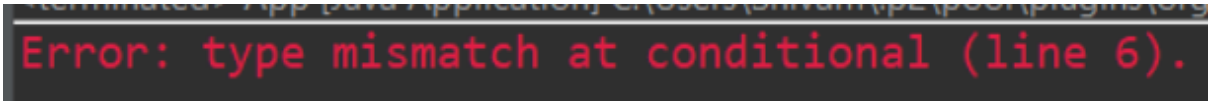
To run the semantic error checking examples (src/tests/semantic), you do not need to choose a test file. You can provide the same file twice through the command line or file prompt.

bool_function_semantics.txt



At line 6, there is a function with return type BOOL. However, the function's return statement at line 7 returns an INT. On line 10, the function with BOOL return type is missing the return statement.

conditional_semantics.txt

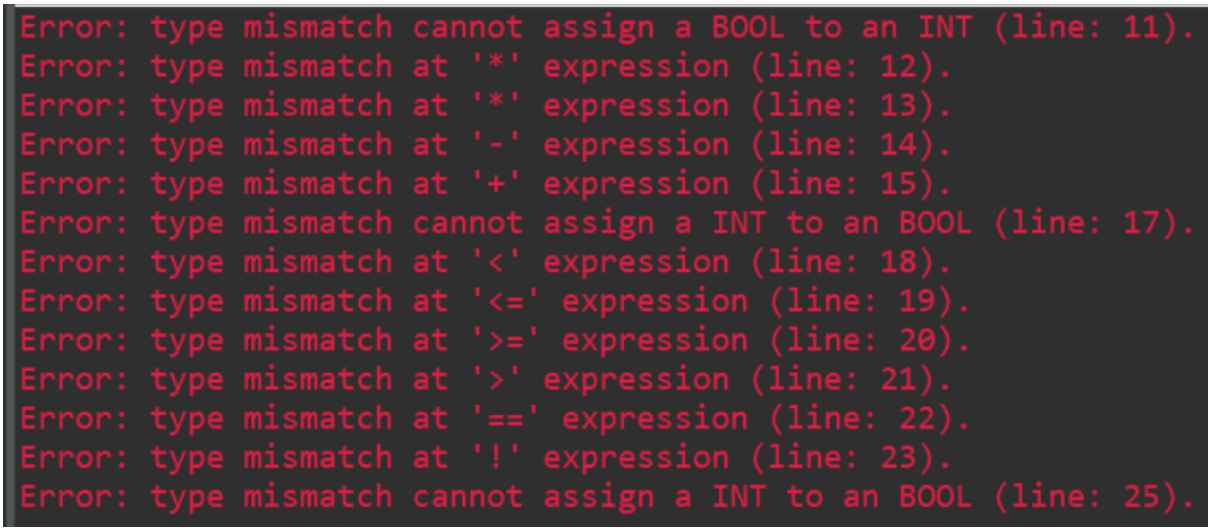


At line 6, there is an if statement with a conditional that is of type INT. Conditionals must evaluate to a BOOL, so using an integer variable is not semantically correct.

duplicate_function.txt

At line 6, we are creating another function with a duplicate name. All function names should be unique, our language does not support method overloading.

expression_semantics.txt



Each error is related to assigning a value with the wrong data type to INT and BOOL variables. It thoroughly checks all operations and a function call with INT return type.

function_call_semantics.txt

Each error is related to sending in arguments with types that do not match the types of the parameters of the function. It thoroughly checks all possible combinations of input.

int_function_semantics.txt

```
Error: type mismatch function 'foo' must return an int (line 6).
Error: function 'bar' with return type INT, must have return statement (line 10).
```

At line 6, there is a function with return type INT. However, the function's return statement at line 7 returns a BOOL. On line 10, the function with integer return type is missing a return statement.

main_function_semantics.txt

```
Error: function 'main' cannot have any parameters.
Error: function 'main' cannot have a return type.
```

The main function should have no parameters or a return type.

main_function_missing.txt

```
Error: program is missing function 'main'.
```

Program does not have a main function.

undeclared_variable.txt

```
Error: variable 'b' not declared (line: 5).
```

At line 5, there is a reference to a variable that has not been declared.

6. Limitations

- Functions can only have one return statement at the end if they return anything.
- Test language has no semantic error checking and it does not support all features supported by the programming language such as conditionals, variable assignment/declaration, print, recursion, etc…
- Test functions cannot call other test functions.