Team Compiler Project
Project Report

CMPE 152 Sec 01
Professor Ronald Mak

Team Peace
Jasper Favis
Katrina Dresser
Shalvin Prasad
Nishin Shouzab

Peace

We created a language called Peace which is a mixture of Pascal and C++. The basic program structure is similar to Pascal while the syntax is more similar to C++, in an attempt to create a straightforward and easy to use language.

```
PROG PeaceMathTestProgram                                   Program Header

    integer - nTest, nfTest, aTest, bTest, gcdTest;         Program Variable Declarations

    integer factorial(integer n)
    {
        integer i, nf;
        i = 1;
        nf = i;
        WHILE(i <= n)
        {
            nf = nf * i;
            i = i + 1;
        };

        return nf
    };                                                      Function and Procedure Definitions

    integer gcd(integer a, integer b)
    {
        WHILE(a != b)
        {
            IF a > b
                a = a - b;
            IF a <= b
                b = b - a;
        };
        return a
    };

MAIN                                                        Begin Main
{
    nTest = 5;
    aTest = 9;
    bTest = 15;

    nfTest = factorial(nTest);
    PRINT("%d! = %d\n", nTest, nfTest);

    gcdTest = gcd(aTest, bTest);
    PRINT("the gcd of %d and %d is %d\n", aTest, bTest, gcdTest);
}                                                           End Main
ENDPROG                                                     Program End
```

Figure 1. Sample Peace Program, Emphasis on Program Structure

Language Constructs

We implemented the following language constructs:

- Statements: assignment, if, while, print, procedure calls
- Expressions: multiply/ divide, add/ subtract, number, signed number, variable, boolean, function calls, parenthetical
- Functions and procedures with pass by value parameters

Generated Jasmin Object Code

The following figures illustrate examples of language constructs and the generated Jasmin code that is produced.

Procedure Definition

In the example procedure below, two integers are passed as parameters and two local variables of type real are declared. In line 27, the procedure signature shows the procedure name, followed by

the parameter types, (two integers) and the return type (void). The parameters are the first local variables added, with *a* stored in slot 0 and *b* stored in slot 1, as shown on lines 28 and 29. Variables *c* and *d* are stored into slots 2 and 3 as shown in line 33 and 34. In total, there are four local variables, so the local stack has a size of 4 which is emitted on line 53. Since *a* and *b* are local variables, the instruction, *iload,* is used to push *a* and *b* onto the stack; since they are integers, the instruction, *iadd,* is used to add them and push the result to the stack. The result is stored to *alpha* which is a program variable, so the instruction, *putstatic* is used to pop the result into *alpha.* The float equivalent instructions are used to add *c* and *d,* and store the result into the program variable, *beta.*

```
 1 PROG Test
 2
 3     integer - alpha;
 4     real - beta;
 5
 6     VOID add(integer a, integer b)
 7     {
 8         real c, d;
 9
10         alpha = a + b;
11         beta  = c + d;
12     };
13
14 MAIN {
15
16     add(3,4);
17
18 }ENDPROG
```

```
27 .method private static add(II)V
28     .var 0 is a I
29     .var 1 is b I
30
31 ; realc,d
32
33     .var 2 is c F
34     .var 3 is d F
35
36 ; alpha=a+b
37
38     iload 0
39     iload 1
40     iadd
41     putstatic          Test/alpha I
42
43 ; beta=c+d
44
45     fload 2
46     fload 3
47     fadd
48     putstatic          Test/beta F
49
50 ;
51
52     return
53 .limit locals 4
54 .limit stack 2
55 .end method
```

Figure 2. Generated Objected Code for an Example Procedure Definition

Function Definition

The example function below returns the sum of two integers that are passed as parameters. Since there are no other declared local variables, the local stack has a size of 2, which is shown on line 40. The only statement in the function is the return statement which returns the sum, so *a* and *b* are loaded onto the stack with *iload* and added with *iadd*; instead of storing into a variable, the instruction *ireturn* is called leaving the result on the stack.

```
PROG Test

    integer - alpha;
    real - beta;

    integer add(integer a, integer b)
    {
        return a + b
    };

MAIN {

    alpha = add(3,4);

}ENDPROG
```

```
.method private static add(II)I
    .var 0 is a I
    .var 1 is b I

;


; returna+b

    iload 0
    iload 1
    iadd
    ireturn
.limit locals 2
.limit stack 2
.end method
```

Figure 3. Generated Object Code for an Example Function Definition

Procedure Call

The example procedure explained previously is called in the *MAIN* with two integer constants. In the Jasmin code, the two constants, 3 and 4, are pushed to the stack, and popped when the procedure, *add* is called with *invokestatic.*

```
PROG Test

    integer - alpha;
    real - beta;

    VOID add(integer a, integer b)
    {
        real c, d;

        alpha = a + b;
        beta  = c + d;
    };

MAIN {

    add(3,4);

}ENDPROG
```

```
; add(3,4)

    ldc             3
    ldc             4
    invokestatic    Test/add(II)V

;
```

Figure 4. Generated Object Code for an Example Procedure Call

Function Call

The function, *add* is called and the result is stored into the program variable, *alpha.* Similar to the procedure, the function is called with constants 3 and 4 as the parameters, so 3 and 4 are pushed to the stack with the instruction *ldc,* and the function is called with *invokestatic.* When the function returns, the sum of *a* and *b* is left on the stack, and the instruction *putstatic* is called to store the result into alpha.

```
1 PROG Test
2
3     integer - alpha;
4     real - beta;
5
6     integer add(integer a, integer b)
7     {
8         return a + b
9     };
10
11 MAIN {
12
13     alpha = add(3,4);
14
15 }ENDPROG
```

```
55 ; alpha=add(3,4)
56
57     ldc             3
58     ldc             4
59     invokestatic    Test/add(II)I
60     putstatic       Test/alpha I
61
```

Figure 5. Generated Object Code for an Example Function Call

If Statement / Boolean Expression (integer compare)

Our compiler emits Jasmin code for the following *IF* statement and *boolean* expression based on the code templates in *Figures 6* and *7*. Lines 48 to 55 in *Figure 8* contain the Jasmin code for comparing two integer expressions. In the example program, *Test,* the values to be compared reside in the program field variable *alpha* and the *MAIN* method's local variable *beta*. As shown on lines 48 and 49, the correct Jasmin load instructions push the integer values onto the operand stack, followed by the correct integer compare-and-branch instruction.

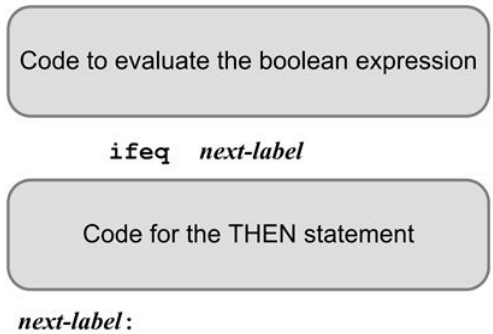

Figure 6. Code Template for *IF* Statement

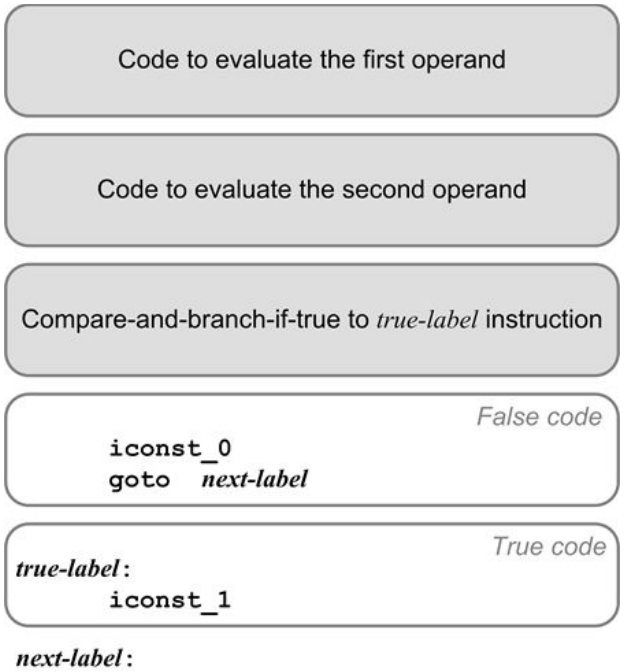

Figure 7. Code Template for *Boolean* Expression (Integer)

```
1 PROG Test
2
3     integer - alpha;
4
5 MAIN {
6
7     integer beta;
8
9     alpha = 2;
10    beta  = 10;
11
12    IF ( alpha != beta )
13    {
14        alpha = alpha + 8;
15    }
16
17 }ENDPROG
```

```
48      getstatic          Test/alpha I
49      iload 0
50      if_icmpne          L001
51      iconst_0
52      goto               L002
53 L001:
54      iconst_1
55 L002:
56      ifeq               L003
57
58 ; {alpha=alpha+8;}
59
60
61 ; alpha=alpha+8
62
63      getstatic          Test/alpha I
64      ldc                8
65      iadd
66      putstatic          Test/alpha I
67
68 ;
69
70 L003:
```

Figure 8. Generated Object Code for an Example IF Statement

While Statement / Boolean Expression (float compare)

The emitted Jasmin code for the *WHILE* statement follows the code template in *Figure 9* and uses the same code template in *Figure 7* for the boolean expression. The emitted code on lines 40 to 52 in *Figure 10* compares two floats using the less-than-or-equal operation. Since float comparisons require the *fcmp* instruction which can only determine whether the values are equal,

less-than, or greater-than, additional instructions were needed to handle the rest of the compare operations. After the two float values are pushed onto the operand stack as shown in lines 40 and 41, the *fcmp* instruction pops the operands, compares them, and pushes an integer value representing one of the relationships mentioned earlier. The instructions *iflt* and *ifeq* each need the result of *fcmp* to determine whether the operands represent a less-than or equal relationship. Since *fcmp* only pushes one value onto the stack, the instruction, *dup*, was emitted to duplicate the value so that both *iflt* and *ifeq* may use it. However, if *iflt* evaluates to true, there is no need to evaluate the second instruction, *ifeq*, and thereby use the duplicate value on the stack, so the program branches to label *L002* and pops it off. If *iflt* evaluates to false, *ifeq* is executed and uses the duplicate value. Unfortunately, the program was unable to run due to several unknown syntax errors involving the instructions *dup* and *iflt*.

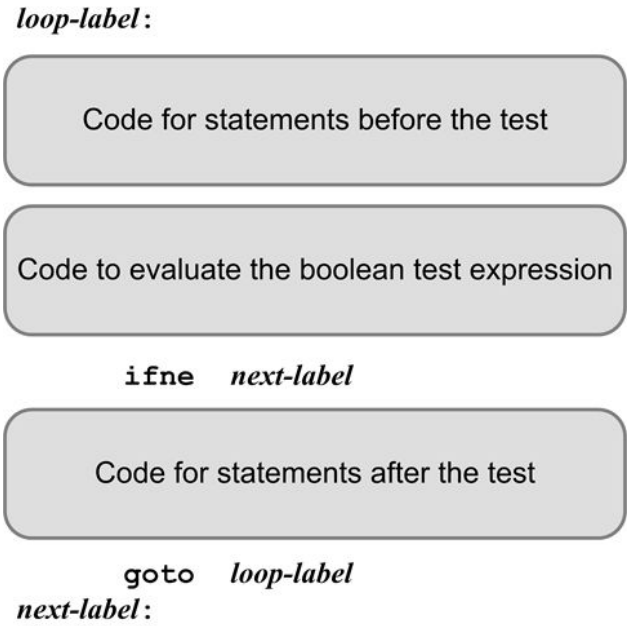

Figure 9. Code Template for While Statement

```
1 PROG Test
2
3     real - alpha;
4
5 MAIN {
6
7     alpha = 0;
8
9     WHILE ( alpha <= 28.5 )
10    {
11        alpha = alpha + 3.5625;
12    }
13
14 }ENDPROG
```

```
39 L001:
40     getstatic          Test/alpha F
41     ldc                28.5
42     fcmp
43     dup
44     iflt               L002
45     ifeq               L003
46     iconst_0
47     goto               L004
48 L002:
49     pop
50 L003:
51     iconst_1
52 L004:
53     ifeq               L005
54
55 ; {alpha=alpha+3.5625;}
56
57
58 ; alpha=alpha+3.5625
59
60     getstatic          Test/alpha F
61     ldc                3.5625
62     fadd
63     putstatic          Test/alpha F
64
65 ;
66
67     goto               L001
68 L005:
69
```

Figure 10. Generated Object Code for an Example While Statement

Print Statement

Our print statement is able to handle a variable number of parameters to be formatted following the literal string constant. To handle printing with no format parameters, we invoke the *print* function instead of the *printf* function. In the example below, the print statement prints one local variable from the *MAIN,* and two program variables. On line 60, the *iload* instruction is used for the first format parameter and local variable, *one.* The second and third format parameters are program variables, so the instruction *getstatic* is used, shown on lines 65 and 70.

```
1 PROG Test
2
3     integer - two, three;
4
5 MAIN {
6
7     integer one;
8
9     one   = 1;
10    two   = 2;
11    three = 3;
12
13    PRINT ("ONE is %d\nTWO is %d\nTHREE is %d", one, two, three);
14
15 }ENDPROG
```

```
54    getstatic         java/lang/System/out Ljava/io/PrintStream;
55    ldc               "ONE is %d\nTWO is %d\nTHREE is %d"
56    iconst_3
57    anewarray         java/lang/Object
58    dup
59    iconst_0
60    iload_0
61    invokestatic      java/lang/Integer.valueOf(I)Ljava/lang/Integer;
62    aastore
63    dup
64    iconst_1
65    getstatic         Test/two I
66    invokestatic      java/lang/Integer.valueOf(I)Ljava/lang/Integer;
67    aastore
68    dup
69    iconst_2
70    getstatic         Test/three I
71    invokestatic      java/lang/Integer.valueOf(I)Ljava/lang/Integer;
72    aastore
73    invokevirtual     java/io/PrintStream.printf(Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;
74    pop
```

Figure 11. Generated Object Code for an Example Print Statement

Building and Running the Compiler

We created and tested two main sample programs, *PeaceTestProgram.peace* and *PeaceMathTestProgram.peace.* The following is a description of each program, and the steps we took to run each program.

General settings/files to check:

- ANTLR 4.7.2 is used and not version 4.4 (we had a recurring problem of the ANTLR Tool reverting to version 4.4)
- Regenerate the grammar file by right-clicking on the grammar file ->*Run As* -> *Generate ANTLR Recognizer.*

PeaceTestProgram.peace

This test program prints out *n* number of diamonds of size (width) *s* to the console. Three functions were written; one to print the top half of the diamond, one to print the bottom half of the diamond, and one to print a full diamond, which calls the two previous functions. To specify the size and number of diamonds, just change the values of *n* and *s* in the *MAIN,* which are the values passed to the function, *fullDiamond(integer s, integer n).*

1. To build the compiler, first check that the argument in *Run Configurations* is *PeaceTestProgram.peace.*
2. Select the project in the *Package Explorer* and build the project
3. Verify that the cross reference table is printed in the console, no errors, and that the Jasmin file was created.
4. Copy the Jasmin file, *PeaceTestProgram.j* into the *jasmin-2.4* Ubuntu directory, which should also contain the *PascalRTL.jar* file.
5. Open terminal and change to the *jasmin-2.4* directory.
6. Run the command `java -jar jasmin.jar PeaceTestProgram.j` . Verify that *PeaceTestProgram.class* file was generated.
7. Run the command `java -cp .:PascalRTL.jar PeaceTestProgram` . Our sample program initially tests *n=5* and *s=12,* so the output in the terminal should print 5 stars in a row, each star with a width of 11 '*'.

```
katrina@Ubuntu-18:~/jasmin-2.4$ java -jar jasmin.jar PeaceTestProgram.j
Generated: PeaceTestProgram.class
katrina@Ubuntu-18:~/jasmin-2.4$ java -cp .:PascalRTL.jar PeaceTestProgram
     *          *          *          *          *
    ***        ***        ***        ***        ***
   *****      *****      *****      *****      *****
  *******    *******    *******    *******    *******
 *********  *********  *********  *********  *********
***********************************************************
 *********  *********  *********  *********  *********
  *******    *******    *******    *******    *******
   *****      *****      *****      *****      *****
    ***        ***        ***        ***        ***
     *          *          *          *          *


                0.04 seconds total execution time.
```

Figure 12. Output of *PeaceTestProgram.peace*

PeaceMathTestProgram.peace

This test program performs two math computations, which are each implemented in a function. The first function computes a factorial using a while loop, and the second function computes the greatest common divisor (GCD) of two numbers. The main program calls both functions and prints a statement for each computation and result.

1. To build the compiler, first check that the argument in *Run Configurations* is *PeaceMathTestProgram.peace.*
2. Select the project in the *Package Explorer* and build the project
3. Verify that the cross reference table is printed in the console, no errors, and that the Jasmin file was created.
4. Copy the Jasmin file, *PeaceMathTestProgram.j* into the *jasmin-2.4* Ubuntu directory, which should also contain the *PascalRTL.jar* file.
5. Open terminal and change to the *jasmin-2.4* directory.

6. Run the command `java -jar jasmin.jar PeaceTestProgram.j` . Verify that the *PeaceMathTestProgram.class* file was generated.
7. Run the command `java -cp .:PascalRTL.jar PeaceTestProgram` . Our sample program tests 5 factorial and computes the GCD of 9 and 15. The output in the terminal should print "5! = 120" followed by "the gcd of 9 and 15 is 3" on the next line.

```
katrina@Ubuntu-18:~/jasmin-2.4$ java -jar jasmin.jar PeaceMathTestProgram.j
Generated: PeaceMathTestProgram.class
katrina@Ubuntu-18:~/jasmin-2.4$ java -cp .:PascalRTL.jar PeaceMathTestProgram
5! = 120
the gcd of 9 and 15 is 3

                0.03 seconds total execution time.
```

Figure 13. Output of *PeaceMathTestProgram.peace*