

CMPE 152: Compiler Design

November 7 Class Meeting

Department of Computer Engineering
San Jose State University



Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak



Code Templates

- Syntax diagrams
 - Specify the source language grammar
 - Help us write the parsers

- Code templates
 - Specify what object code to generate
 - Help us write the code emitters

Code Template for a Pascal Program

```
.class public program-name
.super java/lang/Object
```

Program header

Code for fields

```
.method public <init>()V
```

Class constructor

```
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
```

```
.limit locals 1
.limit stack 1
.end method
```

Code for methods

Code for the main method

- ❑ Translate a Pascal program into a public class.
- ❑ Program variables become class fields.
- ❑ Must have a default constructor.
- ❑ Each procedure or function becomes a private **static** method.
- ❑ The main program code becomes the public **static** main method.

Compilation Strategy

- We'll compile a Pascal program as if it were a public Java class.
 - The Pascal program name becomes the Java class name.
- The main program becomes the main method of the Java class.
- We'll compile each program variable as if it were a field of the class.
 - Fields do have names in a Jasmin program.
 - Recall that local variables and parameters are referred to only by their slot numbers.

Compilation Strategy, *cont'd*

- ❑ We'll compile each Pascal procedure or function as if it were a private static method of the Java class.
- ❑ Local variables and formal parameters of the method do not have names in a Jasmin program.
- ❑ Jasmin instructions refer to local variables and parameters by their slot numbers of the local variables array.

Jasmin Type Descriptors

Java Scalar type	Jasmin Type Descriptor
int	I
float	F
boolean	Z
char	C

Java Class	Jasmin Type Descriptor
java.lang.String	Ljava/lang/String;
java.util.HashMap	Ljava/util/HashMap;
Newton	LNewton;

Java Array type	Jasmin Type Descriptor
java.lang.String[]	[Ljava/lang/String;
Newton[][]	[[LNewton;
int[][][]	[[[I;

Program Fields

```
.class public program-name  
.super java/lang/Object
```

Program header

Code for fields

```
.method public <init>()V  
  
    aload_0  
    invokenonvirtual java/lang/Object/<init>()V  
    return  
  
.limit locals 1  
.limit stack 1  
.end method
```

Class constructor

Code for methods

Code for the main method



Program Fields, *cont'd*

□ For example:

```
PROGRAM test;  
VAR  
    i, j, k : integer;  
    x, y    : real;  
    p, q    : boolean;  
    ch      : char;  
    index   : 1..10;
```

} Pascal program variables

□ Compiles to:

```
.field private static _runTimer LRunTimer;  
.field private static _standardIn LPascalTextIn;  
.field private static ch C  
.field private static i I  
.field private static index I  
.field private static j I  
.field private static k I  
.field private static p Z  
.field private static q Z  
.field private static x F  
.field private static y F
```

Classes **RunTimer** and **PascalTextIn** are defined in the **Pascal Runtime Library** **PascalRTL.jar** which contains runtime routines written in Java.

Code Template for the Main Method, *cont'd*

```
.class public program-name  
.super java/lang/Object
```

Program header

Code for fields

```
.method public <init>()V  
  
    aload_0  
    invokenonvirtual java/lang/Object/<init>()V  
    return  
  
.limit locals 1  
.limit stack 1  
.end method
```

Class constructor

Code for methods

Code for the main method



Code Template for the Main Method, *cont'd*

Main method header

```
.method public static main([Ljava/lang/String;)V
```

Main method prologue

```
new          RuntimeR
dup
invokenonvirtual RuntimeR/<init>()V
putstatic    program-name/_runtimeR LRuntimeR;
new          PascalTextIn
dup
invokenonvirtual PascalTextIn/<init>()V
putstatic    program-name/_standardIn LPascalTextIn;
```

Code for structured data allocations

Code for compound statement

Main method epilogue

```
getstatic    program-name/_runtimeR LRuntimeR;
invokevirtual RuntimeR.printElapsedTime()V

return

.limit locals n
.limit stack m
.end method
```

- The main method prologue initializes the runtime timer `_runtimeR` and the standard input `_standardIn` fields.
- The main method epilogue prints the elapsed run time.
 - `.limit locals`
`.limit stack`
specify the size of the local variables array and the maximum size of the operand stack, respectively.

Loading a Program Variable's Value

- To load (push) a program variable's value onto the operand stack:

getstatic *program-name/variable-name* *type-descriptor*

- Examples:

`getstatic Test/count I`
`getstatic Test/radius F`

Java Scalar type	Jasmin Type Descriptor
int	I
float	F
boolean	Z
char	C

Storing a Program Variable's Value

- To store (pop) a value from the operand stack into a program variable:

putstatic *program-name/variable-name type-descriptor*

- Examples:

```
putstatic Test/count I
putstatic Test/radius F
```

Java Scalar type	Jasmin Type Descriptor
int	I
float	F
boolean	Z
char	C

Code for Procedures and Functions

```
.class public program-name  
.super java/lang/Object
```

Program header

Code for fields

```
.method public <init>()V  
  
    aload_0  
    invokenonvirtual java/lang/Object/<init>()V  
    return  
  
.limit locals 1  
.limit stack 1  
.end method
```

Class constructor

Code for methods



Code for the main method

Code for Procedures and Functions

Routine header

```
.method private static signature return-type-descriptor
```

Code for local variables

Code for structured data allocations

Code for compound statement

Code for return

Routine epilogue

```
.limit locals n  
.limit stack m  
.end method
```

- Each a private static method.
- Method signature:
 - Routine's name
 - Type descriptors of the formal parameters.
- Example:

```
TYPE  
    arr = ARRAY [1..5] OF real;  
  
FUNCTION func(i, j : integer;  
              x, y : real;  
              p : boolean;  
              ch : char;  
              vector : arr;  
              length : integer)  
    : real;
```

- Compiles to:

```
.method private static func(IIFZC[FI)F
```

Compiling Local Variables

TYPE

```
arr = ARRAY [1..5] OF real;
```

```
FUNCTION func(i, j : integer;  
             x, y : real;  
             p : boolean;  
             ch : char;  
             vector : arr;  
             length : integer)  
: real;
```

VAR

```
n : integer;  
z : real;  
w : arr;
```

□ Compiles to:

```
.method private static func(IIFFZC[FI)F  
.var 5 is ch C  
.var 0 is i I  
.var 1 is j I  
.var 7 is length I  
.var 8 is n I  
.var 4 is p Z  
.var 6 is vector [F  
.var 10 is w [F  
.var 2 is x F  
.var 3 is y F  
.var 9 is z F  
.var 11 is func F
```

Routine header

.method private static signature return-type-descriptor

Code for local variables

Code for structured data allocations

Code for compound statement

Code for return

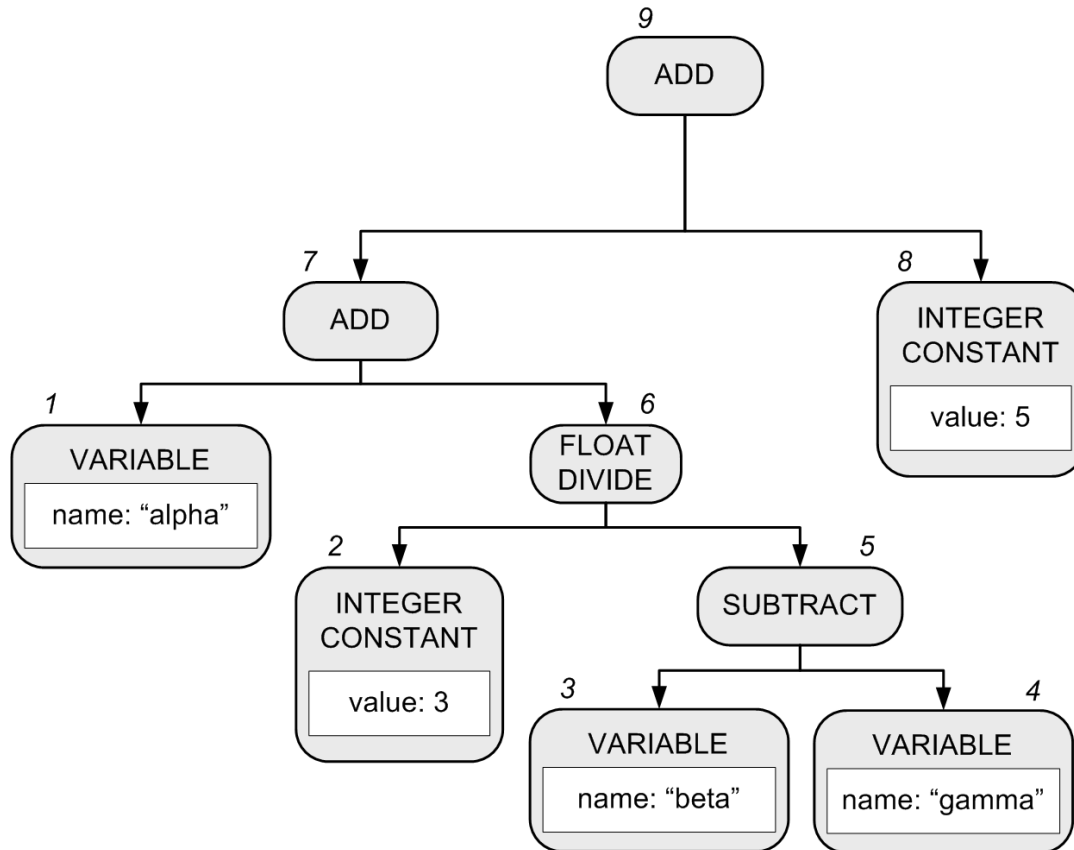
Routine epilogue

```
.limit locals "  
.limit stack "  
.end method
```

Add a **slot number** for the local variables array to each variable's symbol table entry.

Generating Code for Expressions

$\alpha + 3 / (\beta - \gamma) + 5$



□ Recall that in our Pascal interpreter, the expression executor does a postorder traversal of the expression parse tree.

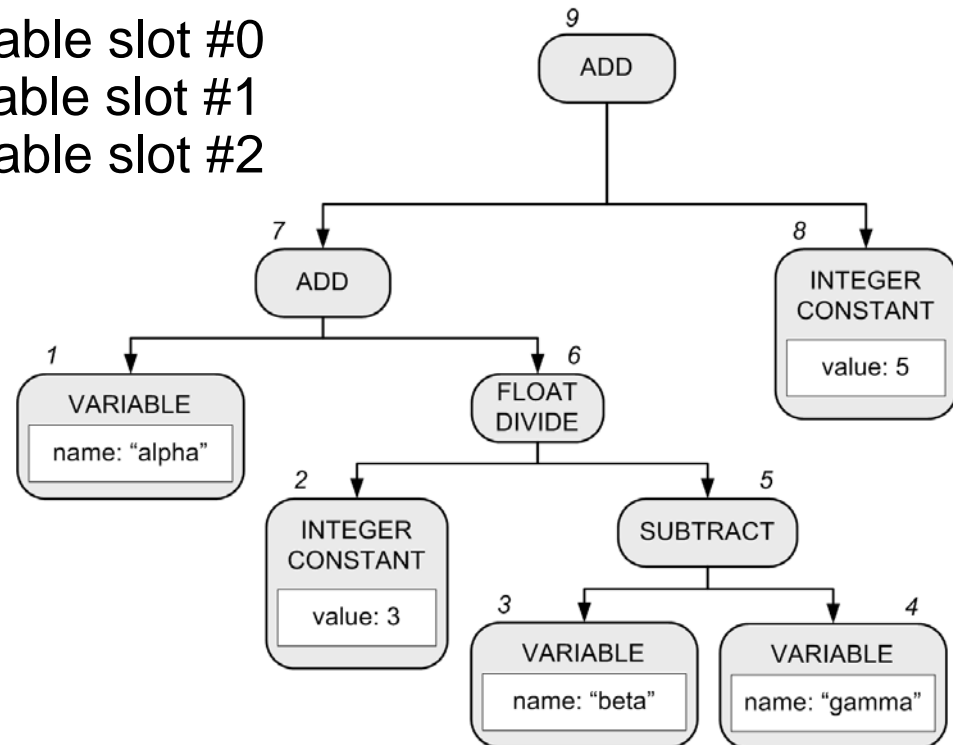
■ Pascal's operator precedence rules are encoded in the structure of the parse tree.

Generating Code for Expressions

- A compiler's expression code generator also does a postorder traversal to generate code.
- Assume that **alpha**, **beta**, and **gamma** are local real variables
- **alpha** → local variable slot #0
- **beta** → local variable slot #1
- **gamma** → local variable slot #2

- Generated code:

```
1 fload_0
2 ldc 3.0
3 fload_1
4 fload_2
5 fsub
6 fdiv
7 fadd
8 ldc 5.0
9 fadd
```



$\alpha + 3/(\beta - \gamma) + 5$

Key Points

- Pascal program
 - ➔ public Jasmin class
- Pascal program (level 1) variables
 - ➔ private static Jasmin class fields (with names)
- Pascal procedure or function
 - ➔ private static Jasmin method
- Pascal procedure or function local variables and formal parameters
 - ➔ local variables array slot numbers (no names)

Tips

- ❑ Write special code emitters for loading (pushing) values onto the operand stack.
- ❑ If loading constants:
 - Determine whether you can emit a shortcut instruction.

Tips, *cont'd*

- ❑ If loading variables:
 - Determine whether it's a program variable (emit a **getstatic** instruction with the field name) or a local variable (emit a load instruction with the slot number).
 - Determine whether you can emit a shortcut instruction for a local variable.
- ❑ Similarly, write special code emitters for storing (popping) values off the operand stack into variables.

What Would James Gosling Do?

- What Jasmin code should you generate?

```
public class Test
{
    private static float test()
    {
        float alpha    = 0; /* slot #0
        float beta     = 10; /* slot #1
        float gamma    = 20; /* slot #2
        int    thirty  = 30; /* slot #3
        int    forty   = 40; /* slot #4
        int    fifty  = 50; /* slot #5

        if (forty == fifty) {
            return alpha + 3/(beta - gamma) + 5;
        }
        else {
            return alpha + thirty/(beta - gamma) + fifty;
        }
    }
}
```

What Would James Gosling Do, *cont'd*

- Run `javap` to disassemble the `.class` file:
`javap -l -p -s -c Test.class`

```
return alpha + 3/(beta - gamma) + 5;
```

```
return alpha + thirty/(beta - gamma) + fifty;
```

```
fload_0
```

```
ldc      3.000000
```

```
fload_1
```

```
fload_2
```

```
fsub
```

```
fdiv
```

```
fadd
```

```
ldc      5.000000
```

```
fadd
```

```
freturn
```

```
fload_0
```

```
iload_3
```

```
i2f
```

```
fload_1
```

```
fload_2
```

```
fsub
```

```
fdiv
```

```
fadd
```

```
iload    5
```

```
i2f
```

```
fadd
```

```
freturn
```

```
float alpha = 0; /* #0  
float beta  = 10; /* #1  
float gamma = 20; /* #2  
int  thirty = 30; /* #3  
int  forty  = 40; /* #4  
int  fifty  = 50; /* #5
```

What Would James Gosling Do, *cont'd*

- ❑ If you have a construct in your source language and you don't know what Jasmin code to generate for it:
 - Write the equivalent construct in Java.
 - Compile the Java into a `.class` file.
 - Run `javap` to get an idea of what code you should generate.
- ❑ Unfortunately, `javap` output is not compatible with the Jasmin assembler.
 - Jasmin won't assemble `javap`-generated programs.

Jasper

- ❑ Also check out the Jasper disassembler:
<http://www.angelfire.com/tx4/cus/jasper/>
- ❑ They claim to disassemble **.class** files into Jasmin files suitable for the Jasmin assembler.
 - Jasper is a Java program.
- ❑ Google “Java byte code viewer” for others.

Comparing Integer Values

- Jasmin has a set of instructions each of which compares the top two integer values on the operand stack and then branches if the comparison is true.

Instruction	Action
<code>if_icmpeq label</code>	Branch to <i>label</i> if $[TOS-1] == [TOS]$
<code>if_icmpne label</code>	Branch to <i>label</i> if $[TOS-1] != [TOS]$
<code>if_icmpgt label</code>	Branch to <i>label</i> if $[TOS-1] > [TOS]$
<code>if_icmpge label</code>	Branch to <i>label</i> if $[TOS-1] \geq [TOS]$
<code>if_icmplt label</code>	Branch to <i>label</i> if $[TOS-1] < [TOS]$
<code>if_icmple label</code>	Branch to <i>label</i> if $[TOS-1] \leq [TOS]$

Comparing Integer Values, *cont'd*

Instruction	Action
<code>if_icmpeq label</code>	Branch to <i>label</i> if [TOS-1] == [TOS]
<code>if_icmpne label</code>	Branch to <i>label</i> if [TOS-1] != [TOS]
<code>if_icmpgt label</code>	Branch to <i>label</i> if [TOS-1] > [TOS]
<code>if_icmpge label</code>	Branch to <i>label</i> if [TOS-1] >= [TOS]
<code>if_icmplt label</code>	Branch to <i>label</i> if [TOS-1] < [TOS]
<code>if_icmple label</code>	Branch to <i>label</i> if [TOS-1] <= [TOS]

- The two values are popped off the operand stack.
 - [TOS] is the value at the top of the stack.
 - [TOS-1] is the value just under the one at the top of the stack.

Comparing Integer Values, *cont'd*

- You can also simply compare the single integer value at the top of the operand stack to 0 and then branch if the comparison is true.

Instruction	Action
<code>ifeq label</code>	Branch to <i>label</i> if [TOS] == 0
<code>ifne label</code>	Branch to <i>label</i> if [TOS] != 0
<code>ifgt label</code>	Branch to <i>label</i> if [TOS] > 0
<code>ifge label</code>	Branch to <i>label</i> if [TOS] >= 0
<code>iflt label</code>	Branch to <i>label</i> if [TOS] < 0
<code>ifle label</code>	Branch to <i>label</i> if [TOS] <= 0

- The top value is popped off the stack.

Comparing Other Values

- Instructions **lcmp**, **fcmp**, and **dcmp** compare two long, float, or double values at the top of the operand stack.
 - Each pops the top two values off the operand stack and then pushes the integer value **-1**, **0**, or **1** onto the stack.
 - If $[TOS-1] < [TOS]$, push **-1** onto the stack.
 - If $[TOS-1] = [TOS]$, push **0** onto the stack.
 - If $[TOS-1] > [TOS]$, push **1** onto the stack.
- Use instructions **iflt**, **ifeq**, or **ifgt** to test for the **-1**, **0**, or **1**.