sT: A Simple *Turing* Programming Language

Programming Assignment 3

Code Generation

Due Date: 1:20PM, Tuesday, June 20, 2023

Your assignment is to generate code (in Java assembly language) for the sT language. The generated code will then be translated to Java bytecode by a Java assembler.

1 Assignment

Your assignment will be divided into the following parts:

- initialization
- parsing declarations for constants and variables
- code generation for expressions and statements
- code generation for conditional statements and loops
- code generation for procedure calls

1.1 Language Restrictions

In order to keep the code generation assignment simple that we can implement most of the features of the language, only a subset set of sT language will be considered in this assignment:

- No GET statements.
- No declaration or use of arrays.
- No string variables, i.e. no assignments to string variables. Only string constants and string literals are provided for use in PUT statements.

1.2 What to Submit

You should submit the following items:

- your compiler
- a file describing what changes you have to make to your scanner and parser since the previous version you turned in
- Makefile
- test programs

1.3 Java Assembler

The Java Bytecode Assembler (or simply Assembler) is a program that converts code written in ”Java Assembly Language” into a valid Java .class file.

2 Generating Java Assembly Code

This section describes the major pieces (see Section 1) of the translation of sT programs into Java assembly code. This document presents methods for code generation in each piece and gives sample sT along with the Java assembly code. Please note the label numbers in the examples are arbitrarily assigned.

2.1 Initialization

A sT program is translated into a Java class. An empty sT program `example.st`

will be translated into the following Java assembly code

```
class example
{
  method public static void main(java.lang.String[])
  max_stack 15
  max_locals 15
  {
    return
  }
}
```

Consequently, once the file name is read, a declaration of the corresponding class name must be generated. Furthermore, a method `main` that is declared public and static must be generated.

2.2 Declarations for Variables and Constants

Before generating Java assembly commands for sT statements, you have to allocate storage for declared variables and store values of constants.

2.2.1 Allocating Storage for Variables

Variables can be classed into two types: global and local. All variables that are declared inside compound statements are local, while other variables are global.

Global Variables

Global variables will be modeled as fields of classes in Java assembly language. Fields will be declared right after class name declaration. Each global variable `var` will be declared as a static field by the form

```
  field static type var < = const_value >
```

where `type` is the type of variable `var`, and < = const_value > is an optional initial constant value. For example,

```
  var a :int
  var b := 10
  var c :int
```

will be translated into the following declaration statements in Java assembly

```
  field static int a
  field static int b = 10
  field static int c
```

Local Variables

Local variables in sT will be translated into local variables of methods in Java assembly. Unlike fields (i.e. global variables of sT), local variables will not be declared explicitly in Java assembly programs. Instead, local variables will be numbered and instructions to reference local variables take an integer operand indicating which variable to use. In order to number local variables, symbol tables should maintain a counter to specify "the next available number". For example, consider the following program fragment:

```
begin
  var i :int
  var j :int
  begin
    var k :int
  end
  begin
    var i :int
    var j :int
    var k :int
  end
end
```

the symbol table information will be

```
entering block, next number 0
  i = 0, next number 1
  j = 1, next number 2
entering block, next number 2
  k = 2, next number 3
leaving block, symbol table entries:
  <"k", variable, integer, 2>
entering block, next number 3
  i = 3, next number 4
  j = 4, next number 5
  k = 5, next number 6
leaving block, symbol table entries:
  <"i", variable, integer, 3>
  <"j", variable, integer, 4>
  <"k", variable, integer, 5>
leaving block, symbol table entries:
  <"i", variable, integer, 0>
  <"j", variable, integer, 1>
```

In addition, if an initialization value is given, statements must be generated to put the value onto the operand stack and then store it to the local variable. For instance, if the last statement of the above example is changed to

```
  var k :int := 100
```

a store statement must be generated as well:

```
  sipush 100
  istore 2           // local variable number of k is 2
```

2.2.2 Storing Constants in Symbol Table

Constant variables in sT will not be transformed into fields or local variables in Java assembly. The values of constant variables will be stored in symbol tables.

2.3 Expressions and Statements

2.3.1 Expressions

An expression can be either an variable, a constant variable, an arithmetic expression, or a boolean expression.

Variables

Since string variables are not considered in this project and furthermore Java Virtual Machine does not have instructions for boolean, a variable will be loaded to the operand stack by *iload* instruction if it is local or *getstatic* if it is global. Consider the following program fragment

```
var a :int
procedure foo
  var b :int
     := a ...
     := b ...
end foo
```

The translated program will contain the following Java assembly instructions

```
  getstatic int example.a
  ...
  iload 1  /* local variable number of b is 1 */
```

Constants

The instructions to load a constant in Java Virtual Machine is *iconst_value* or *sipush value* if the constant is a boolean or an integer, or *ldc string* is the constant is a string. Consider the following program fragment

```
const a :int := 10
const b :bool := true
const s :string := "string"
  ...
   := a ...
   := b ...
  put s
   := 5 ...
  ...
```

The translated program will contain the following Java assembly instructions

```
sipush 10 /* := a ... */
...
iconst_1 /* := b ... */
...
ldc "string" /* put s */
...
sipush 5 /* := 5 ... */
```

Arithmetic and Boolean Expressions

Once the compiler performs a reduction to an arithmetic expression or a boolean expression, the operands of the operation will already be on the operand stack. Therefore, only the operator will be generated. The following table lists the mapping between operators in sT and corresponding instructions in Java assembly language.

sT Operator	Integer	sT Operator	Boolean
+	iadd	`and`	iand
−	isub	`or`	ior
*	imul	`not`	ixor
/	idiv		
`mod`	irem		
− (neg)	ineg		

Boolean expressions with relational operators will be modeled by a subtraction instruction followed by a conditional jump. For instance, consider $a < b$.

```
    isub  /* a and b are at stack top */
    iflt L1
    iconst_0  /* false = 0 */
    goto L2
L1: iconst_1  /* true = 1 */
L2:
```

The following table summarizes the conditional jumps for each relational operator:

sT relop	if*cond*	sT relop	if*cond*
<	iflt	`<=`	ifle
>	ifgt	`=>`	ifge
=	ifeq	`not=`	ifne

2.3.2 Statements

Assignments *id := expression*

The right side, i.e. *expression*, will be on the operand stack when this production is reduced. As a result, the code to generate is to store the value at stack top in *id*. If *id* is a local variable, then the instruction to store the result is

```
istore 2  /* local variable number is 2 */
```

On the other hand, if *id* is global, then the instruction will be

```
putstatic type example.id
```

where `type` is the type of *id*.

PUT Statements *put expression*

The PUT statements in sT are modeled by invoking the *print* method in *java.io* package using the following format

```
getstatic java.io.PrintStream java.lang.System.out
...  /* compute expression */
invokevirtual void java.io.PrintStream.print(java.lang.String)
```

if the type of *expression* is a string. Types *int* or *boolean* will replace *java.lang.String* if the type of *expression* is integer or boolean.

```
getstatic java.io.PrintStream java.lang.System.out
...  /* compute expression */
invokevirtual void java.io.PrintStream.print(int)
```

SKIP Statements

A SKIP statement will be compiled to the following java assembly code:

```
getstatic java.io.PrintStream java.lang.System.out
invokevirtual void java.io.PrintStream.println()
```

2.4 If Statements

It is fairly simple to generate code for IF and loop statements. Consider this if-then-else statement:

```
if (false) then
  i := 5
else
  i := 10
end if
```

The following code will be generated

```
  iconst_0
  ifeq Lfalse
  sipush 5
  istore 2  /* local variable number of i is 2 */
  goto Lexit
Lfalse:
  sipush 10
  istore 2
Lexit:
```

2.5 Loops

For each loop, a start label is inserted before the first statement and a jump instruction back to the start label will be added at the end of loop. If an optional `exit` statement is encountered, a test will be performed on the boolean expression and a conditional jump will be executed if the condition is met. Consider the following loop

```
  i := 1
  loop
    exit when (i > 10)
    i := i + 1
  end loop
```

The following instructions will be generated:

```
  sipush 1  /* constant 1 */
  istore 1  /* local variable number of i is 1 */
Lbegin:
  iload 1
  sipush 10
  isub
  ifgt Ltrue
  iconst_0
  goto Lfalse
Ltrue:
  iconst_1
Lfalse:
  ifne Lexit
  iload 1
  sipush 1
  iadd
  istore 1
  goto Lbegin
Lexit:
```

A for loop can be translated in the same way as a loop, as a for loop

```
  for i : 1 .. 10
    ...
  end for
```

is equivalent to

```
  i := 1
  loop
    exit when (i > 10)
    ...
    i := i + 1
  end loop
```

2.6 Function Declaration and Invocation

Functions in sT will be modeled by static methods in Java assembly language.

2.6.1 Function Declaration

If n arguments are passed to a static Java method, they are received in the local variables numbered 0 through $n - 1$. The arguments are received in the order they are passed. For example

```
function add(a :int, b :int) :int
  result a+b
end add
```

will be compiled to

```
method public static int add(int, int)
max_stack 15
max_locals 15
{
  iload 0
  iload 1
  iadd
  ireturn
}
```

A procedure is translated in the same way, with the difference that the type of its corresponding method will be **void** and the return instruction will be **return**.

2.6.2 Function Invocation

To invoke a static method, the instruction *invokestatic* will be used. The following function invocation

```
:= add(a, 10) ...
```

will be compiled into

```
...
iload 1  /* local variable number of a is 1 */
sipush 10  /* constant 10 */
invokestatic int example.add(int, int)
...
```

where the first `int` is the return type of the function and the second and last `int` are the types of formal parameters.

3 Implementation Notes

3.1 Local Variable Numbers

Formal parameters of a function are numbered starting from 0. Local variables in the function are placed right after the formal parameters of the function. For example, if a function has n formal parameters, then the parameters are numbered from 0 to $n-1$, while the first local variable will be numbered n.

3.2 Java Virtual Machine

Once an sT program is compiled, the resulted Java assembly code can then be transformed into Java bytecode using the Java assembler *javaa*. The output of *javaa* will be a class file of the generated bytecode, which can be executed on the Java Virtual Machine. For example, if the generated class is *example.class*, then type the following command to run the bytecode

```
% java example
```

The structure of Java Virtual Machine is described in the book "*Java Virtual Machine Specification*", which can be found at `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

Example

Source sT program *example.st*:

```
{%
 % Example with Functions
 %}

% global variables
const a :int := 5
var c :int

% function declaration
function add (a :int, b :int) :int
  result a+b
end add

% main block
c := add(a, 10)
if (c > 10) then
  put -c
else
  put c
end if
put "Hello World"
```

Generated Java assembly program:

```
/*-------------------------------------------------*/
/*              Java Assembly Code                 */
/*-------------------------------------------------*/

class example
{
/* 6: const a :int := 5 */
  field static int c
/* 7: var c :int */
/* 8: */
  method public static int add(int, int)
  max_stack 15
  max_locals 15
  {
/* 10: function add (a :int, b :int) :int */
    iload 0
    iload 1
    iadd
    ireturn
/* 11:   result a+b */
  }
/* 12: end add */
/* 13:  */
  method public static void main(java.lang.String[])
  max_stack 15
  max_locals 15
  {
    sipush 5
    sipush 10
    invokestatic int example.add(int, int)
    putstatic int example.c
/* 15:   c := add(a, 10) */
    getstatic int example.c
    sipush 10
    isub
    ifgt L0
    iconst_0
    goto L1
L0:
    iconst_1
L1:
    ifeq L2
/* 16:   if (c > 10) then */
    getstatic java.io.PrintStream java.lang.System.out
    getstatic int example.c
    ineg
    invokevirtual void java.io.PrintStream.print(int)
/* 17:     put -c */
    goto L3
L2:
/* 18:   else */
    getstatic java.io.PrintStream java.lang.System.out
    getstatic int example.c
    invokevirtual void java.io.PrintStream.print(int)
/* 19:     put c */
L3:
/* 20:  end if */
    getstatic java.io.PrintStream java.lang.System.out
    ldc "Hello World"
    invokevirtual void java.io.PrintStream.println(java.lang.String)
/* 21:   put "Hello World" */
    return
  }
}
```