sD: A Simple D Programming Language

Programming Assignment 3

Code Generation

Due Date: 1:20PM, Wednesday, June 10, 2025

Your assignment is to generate code (in Java assembly language) for the sD 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 sD language will be considered in this assignment:

- No floating-point numbers.
- No READ 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 uses in PRINT statements.

1.2 What to Submit

You should submit the following items:

- your compiler
- a note
- 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 four major pieces (see Section 1) of the translation of sD programs into Java assembly code. This document presents methods for code generation in each piece and gives sample sD along with the Java assembly code. Please note the label numbers in the examples are arbitrarily assigned. In addition, an extra piece will be added to generate code for initialization.

2.1 Initialization

A sD program is translated into a Java class. An empty sD program `example.sd`

```
void main() {
}
```

will be translated into the following Java assembly code

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

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

2.2 Declarations for Variables and Constants

Before generating Java assembly commands for sD statements, you have to allocate storage for declared variables and store values of constants in the symbol table.

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,

```
int a, b = 10;
int c;
```

will be translated into the following declaration statements in Java assembly

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

Local Variables

Local variables in sD will be translated into local variables of methods in Java assembly. Unlike fields (i.e. global variables of sD), 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:

```
{
  int i, j;
  {
    int k;
  }
  {
    int i, j, k;
  }
}
```

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 statement in the second block of the above example is changed to

```
  int k = 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 sD 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

```
int a;
void main()
{
    int b;
      = a ... ;
      = b ... ;
}
```

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* if the constant is a string. Consider the following program fragment

```
const int a = 10;
const bool b = true;
const string s = "string";
void main()
{
  = a ...;
  = b ...;
  print s;
  = 5 ...;
}
```

The translated program will contain the following Java assembly instructions

```
sipush 10      /* = a ...; */
...
iconst_1       /* = b ...; */
...
ldc "string"   /* print 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 sD and corresponding instructions in Java assembly language.

sD Operator	Integer	sD Operator	Boolean
+	iadd	&&	iand
−	isub	||	ior
*	imul	!	ixor
/	idiv	++	iinc
%	irem	−−	iinc
− (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:

sD relop	if$cond$	sD relop	if$cond$
<	iflt	<=	ifle
>	ifgt	=>	ifge
==	ifeq	!=	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*.

PRINT Statements *print expression;*

The PRINT statements in sD 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. Similarly, a PRINTLN statement for an *expression* of the string type will be compiled to the following java assembly code:

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

2.4 If Statements and Loops

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

```
if (false)
  i = 5;
else
  i = 10;
```

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:
```

For each WHILE loop, a label is inserted before the boolean expression, and a test and a conditional jump will be performed after the boolean expression. Consider the following WHILE loop

```
i = 0;
while (i < 10)
  i = i + 1;
```

The following instructions will be generated:

```
  sipush 0  /* constant 0 */
  istore 1  /* local variable number of i is 1 */
Lbegin:
  iload 1
  sipush 10
  isub
  iflt Ltrue
  iconst_0
  goto Lfalse
Ltrue:
  iconst_1
Lfalse:
  ifeq Lexit
  iload 1
  sipush 1
  iadd
  istore 1
  goto Lbegin
Lexit:
```

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

```
  for (i = 0; i < 10; i = i + 1)
    { ... }
```

is equivelent to

```
  i = 0;
  while (i < 10) {
    { ... }
    i = i + 1;
  }
```

Similarly, a for loop

```
  for (i : 1 .. 10)
    { ... }
```

is equivelent to

```
  i = 1;
  while (i <= 10) {
    { ... }
    i = i + 1;
  }
```

2.5 Function Declaration and Invocation

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

2.5.1 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

```
int add(int a, int b)
{
  return a+b;
}
```

will be compiled to

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

If a function is declared without a return value, the type of its corresponding method will be **void** and the return instruction will be **return**. Note that the declaration of `main()` must be compiled to

```
method public static void main(java.lang.String[])
```

2.5.2 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 sD 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/jvms8.pdf`.

Example

Source sD program *example.sd*:

```
int a;
int add (int a, int b)
{
  return a+b;
}

main()
{
  int c;
  a = 5;
  c = add(a, 10);
  if (c > 10)
    print -c;
  else
    print c;
  println "Hello World";
}
```

Generated Java assembly program:

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

class example
{
  field static int a
/* 1: int a; */
  method public static int add(int, int)
  max_locals 15
  max_stack 15
  {
/* 2: int add (int a, int b) */
/* 3: { */
    iload 0
    iload 1
    iadd
    ireturn
/* 4:   return a+b; */
  }
/* 5: } */
/* 6:  */
  method public static void main(java.lang.String[])
  max_locals 15
  max_stack 15
  {
/* 7: main() */
/* 8: { */
/* 9:   int c; */
    sipush 5
    putstatic int example.a
/* 10:   a = 5; */
    getstatic int example.a
    sipush 10
    invokestatic int example.add(int, int)
    istore 0
/* 11:   c = add(a, 10); */
    iload 0
    sipush 10
    isub
    ifgt L0
    iconst_0
    goto L1
L0:
    iconst_1
L1:
```

```
    ifeq L2
/* 12:   if (c > 10) */
    getstatic java.io.PrintStream java.lang.System.out
    iload 0
    ineg
    invokevirtual void java.io.PrintStream.print(int)
/* 13:     print -c; */
    goto L3
L2:
/* 14:   else */
    getstatic java.io.PrintStream java.lang.System.out
    iload 0
    invokevirtual void java.io.PrintStream.print(int)
L3:
/* 15:     print c; */
    getstatic java.io.PrintStream java.lang.System.out
    ldc "Hello World"
    invokevirtual void java.io.PrintStream.println(java.lang.String)
/* 16:   println "Hello World"; */
    return
  }
/* 17: } */
}
```