P3: Code Generation

Compiler Construction
2013-2014

In this third assignment you implement a code generator for the SPL language, based on the well-typed abstract syntax tree provided by the parser and type inferencer that you implemented in the first and second assignment.

1 Preliminaries

The target language of the SPL code generator is the SSM language. Detailed descriptions of the available machine instructions can be found in Appendix B of the lecture notes *"Implementation of Programming Languages"* (available on Blackboard). Chapter 6 of the same lecture notes provides a possible translation scheme to SSM instructions for a functional programming language consisting of expressions with nested local function definitions. Note that the language there does not support tuples and lists. Since it is a functional language, no assignments and sequences of statements are available.

SPL has only one level of functions and hence no complications with nested functions. Hence, the problems with finding the arguments in the stack does not occur. In contrast with the compilation scheme in the given lecture notes, SPL has data types. Especially for lists, it is very unpractical to store them on the stack. The size of the list is in general unknown at compile time, hence it is at compilation time unknown how big the required stack space would be. It is far more convenient to store such values on a heap instead of on a stack. In order to enable the storage of values in the heap we have extended the SSM language with four instructions to manipulate a heap:

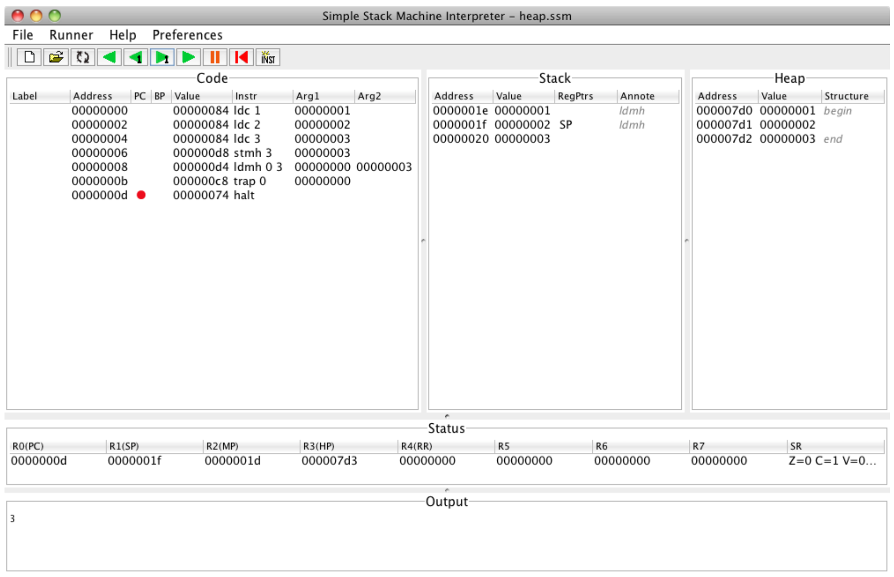
Instruction	Description	Example

`sth`	Store into Heap. Pops 1 value from the stack and stores it into the heap. Pushes the heap address of that value on the stack.	`ldc 5`
		`sth`

`stmh`	Store Multiple into Heap. Pops values from the stack and stores it into the heap, retaining the order of the values. Same as single store variant but the inline parameter is size. Pushes the heap address of the last value on the stack.	`ldc 1`
		`ldc 2`
		`ldc 3`
		`stmh 3`

`ldh`	Load from Heap. Pushes a value pointed to by the value at the top of the stack. The pointer value is offset by a constant offset.	`ldc 5`
		`sth`
		`ldh 0`

`ldmh`	Load Multiple from Heap. Pushes values pointed to by the value at the top of the stack. The pointer value is offset by a constant offset. Same as single load variant but the second inline parameter is size.	`ldc 1`
		`ldc 2`
		`ldc 3`
		`stmh 3`
		`ldmh 0 3`



Code

Label	Address	PC	BP	Value	Instr	Arg1	Arg2
	00000000			00000084	ldc 1	00000001	
	00000002			00000084	ldc 2	00000002	
	00000004			00000084	ldc 3	00000003	
	00000006			000000d8	stmh 3	00000003	
	00000008			000000d4	ldmh 0 3	00000000	00000003
	0000000b			000000c8	trap 0	00000000	
	0000000d	●		00000074	halt		

Stack

Address	Value	RegPtrs	Annote
0000001e	00000001		*ldmh*
0000001f	00000002	SP	*ldmh*
00000020	00000003		

Heap

Address	Value	Structure
000007d0	00000001	*begin*
000007d1	00000002	
000007d2	00000003	*end*

Status

R0(PC)	R1(SP)	R2(MP)	R3(HP)	R4(RR)	R5	R6	R7	SR
0000000d	0000001f	0000001d	000007d3	00000000	00000000	00000000	00000000	Z=0 C=1 V=0...

Output

```
3
```

Figure 1: Screenshot of the extended and improved SSM interpreter

1.1 Simulator

To be able to conveniently test and run your generated instructions, an SSM interpreter (depicted in Figure 1) is available from Blackboard as an executable Java archive file named `ssm.jar`. Note that this is an extended and improved version of the original SSM language and interpreter that did not include a heap. Make sure to always end your generated code with the instruction `trap 0`. This `trap 0` instruction prints the value on top of the stack to the output screen of the interpreter.

2 Data Types

The language treated in the lecture notes at Bb is relatively poor with respect to data types. All values are basic data types. When such a value is used as function argument, the value is simply pushed on the stack. The situation in SPL is more complicated.

First, there are data types such as tuples and lists. The default solution is to build the data structure in the heap, some piece of memory, rather than on the stack. We typically pass a pointer to the data structure in the heap on the stack, rather than the data structure itself.

Second, in SPL there are assignments. This implies that it does matter whether we use the data structure or variable itself, its value, or a copy of some object.

3 Calling Semantics

Having assignments and data structures in the language makes it important to define the semantics of those concepts clearly.

Most languages agree on the semantics of basic values like integers. A variable contains such a value, rather than a pointer to an object containing such a value. A function gets a copy of the value as argument. Function arguments behave identical to variables inside the function body.

For datastructures, objects, there is more variation.

- In C, and hence in C++, a variable is just a name for an object. A function argument behaves like a variable: the argument is just a name for an object. Hence a function argument is a *copy* of the object. When such an object contains references to other objects, like the tail of a list, these other objects are not copied. Just the reference to those objects is copied. In object orientation this is known as a shallow copy.

  You can make those copies of objects on the stack. The type information tells you what objects can be expected. There is a slight complication with lists. The type system does not tell you whether the list is empty or not. Most likely you need less space to store an empty list on the stack. By always reserving space for a non-empty list, the empty list will certainly fit.

- The language C also has an elaborated notion of pointers. One can have pointer variables and pointers as function arguments. Assigning to such a pointer variable just changes the pointer stored in the variable, not the object indicated by this pointer. It is possible to obtain (parts of) the object indicated by such a pointer variable and assign a new value to them. When we use a pointer as function argument, the function gets a copy of the address stored in the pointer object. The object itself is not affected by using it as a function argument. When we change the object via this pointer inside the function this has a global effect.

  We do not recommend to have ordinary variables and function arguments as well as pointer variables and function arguments in SPL. However, interpreting every variable and function argument as a pointer is an option.

- In C one also has *by-reference* variables as function arguments. Those function arguments are marked with a `&`. Those function arguments behave as pointer arguments with implicit dereferencing.

  When you chose to interpret variables as pointer to objects in SPL, implicit dereferencing is most likely the desired interpretation.

- In Java one always uses *by-value* parameter passing and assignment. The variable or function arguments get a copy of the value. However, variables as well as function arguments are references to objects, rather than the objects themselves. Basic values like integers are not objects. Hence, the value is copied in any assignment and used as method argument.

  In this respect Java variables and function arguments behave like pointers in C and values in functional languages.

- Functional languages, like Clean, always pass a pointer to a data structure and have implicit dereferencing. Since there is no assignment this is less critical, it provides just a convenient implementation of lazy evaluation.

  In order to obtain a similar effect in SPL a deep copy of arguments would be required. This will greatly limit the effect of assignments. It is at least a point of discussion whether this is the desired behaviour in an imperative language like SPL.

Until now we have been somewhat vague in the semantics of SPL. In order to implement the translation to SSM the behaviour of variables, assignments and function arguments has to determined clearly. You are free to choose the semantics of SPL to be used in your compiler under two conditions. First, you give a clear description of the semantics to be implemented. It is not required that this is a complete formal semantics, a concise informal or semiformal description suffices. Second, you add some well documented test cases showing that your compiler implements the described semantics for SPL.

3.1 Polymorphism

As a further point of concern SPL has polymorphic functions, like

```
t id(t x)
{
    return x;
}

a K (a x, b y)
{
    return x;
}
```

Also functions like `[t] reverse ([t] list)` are polymorphic. When the compiler generates code for such a function the type of the actual argument is not known. The compiler either has to use an argument handling that works for each and every type of argument, or the compiler has to make various versions for different types of arguments.

In functional languages those polymorphic functions are quite common. In imperative and object oriented languages this somewhat extraordinary. Generic programming in Java and templates in C++ approximate this use of polymorphic arguments.

For code generation it is convenient if only one version is required of the code for each function because this reduces the amount of code to be generated, and we do not have trouble to determine which version of the code for some function has to be used in a given situation. Obviously this imposes some restrictions on the handling of function arguments in the generated code.

4 Memory Management

The creation and deletion of objects in the generated program is called memory management. Again there are interesting choices to be made. In a simple version the deletion of objects is left to the user (as for persistent objects in C). A more advanced solution uses automatic garbage collection as in Java or functional programming languages.

It is fine to use a very simple memory management in the exercise. When a new object is needed it is simply created as long as there is memory available. When the heap is full the program stops with an appropriate message.

In many cases the type of an object referenced to by a pointer tells you how many words in the heap belong to that object. Again, lists are a little more complicated because the type system does not tell whether the list is empty or not. A safe strategy is to reserve space for the head and tail of the list, because then the empty list will certainly fit. The default way to distinguish empty lists from non-empty lists is by adding a *tag*. Such a tag is an additional word in memory containing information about the content of this memory block. In some situations we can fit the tag in unused bits of the ordinary content of the memory reserved for the object. For instance pointers are supposed to be valid heap addresses. Whenever there are more integers than valid words in the heap each pointer has some spare bits.

In a simple memory model each block in memory has the same size. Sometimes this is not necessary.

5 Submission

When submitting your solution, please include the following:

- A document, subsuming the document from the previous assignment, containing:
  - Your full names and student numbers.
  - The chapters from the previous exercise.
  - A chapter describing the chosen semantics of `SPL`.
  - A chapter explaining the compilation schemes used in your compiler. A concise informal or semiformal description suffices, so a formal description of the compilation scheme is welcome, but not required. Typical things to explain here are calling conventions, stack management, stack layout, heap layout, and heap management.
  - A short description of the purpose of the example programs and the test results.
  - A concise but precise description of how the work was divided amongst the members of the team.
- At least ten nontrivial example programs that sufficiently validate your implementation, in separate plain-text `.spl` files.

Please submit your solution on or before **Friday, May 16 2014, 24:00h** on Blackboard. The group seminar in which you present your work will be scheduled on Monday and Tuesday, May 12 and 13.