Phase 3: Code Generation

Compiler Construction
Radboud University

April 25, 2019

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 checker 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"*, subsequently referred to as *the lecture notes*, available on Blackboard. You can also refer to the online documentation of SSM https://www.staff.science.uu.nl/~dijks106/SSM/. Chapter 6 of the 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 functional language in these lecture notes does not support tuples, lists, statements and input/output.

1.1 Heap

SPL has only one level of functions and hence no complications with nested functions. The problems with finding arguments on the stack doesn't occur. In contrast to the language in the lecture notes, SPL has data types. Especially for lists, it is unpractical to store them on the stack. The size of lists is in general unknown at compile time, hence the required stack space is unknown. It is far more convenient to store such values on the heap instead of the stack. SSM has four instructions to manipulate the heap. They are listed in Table 1.

1.2 Interpreter

To be able to conveniently test and run your generated instructions, an SSM interpreter is available in both binary and source code. You can get it here: https://www.staff.science.uu.nl/~dijks106/SSM/download.html

1.3 Traps

The instruction `trap` invokes a system-call determined by its argument. It can be used for input and output. Table 2 lists all available traps.

Make sure to end your generated code with the instruction `trap 0` to print the topmost stack value, unless you have a more fancy form of output implemented.

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`

Table 1: Instructions to manipulate the heap.

Value	Description
0	Pop the topmost element from the stack and print it as an integer.
1	Pop the topmost element from the stack and print it as a unicode character. Since the first 128 unicode characters are the 7-bit ASCII characters, you can also use the plain ASCII characters for output.
10	Ask the user for an integer input and push it on the stack.
11	Ask the user for a unicode character input and push it on the stack.
12	Ask the user for a sequence of unicode characters input and push the characters on the stack terminated by a null-character.
20	Pop a null-terminated file name from the stack, open the file for reading and push a file pointer on the stack.
21	Pop a null-terminated file name from the stack, open the file for writing and push a file pointer on the stack.
22	Pop a file pointer from the stack, read a character from the file pointed to by the file pointer and push the character on the stack.
23	Pop a character and a file pointer from the stack, write the character to the file pointed to by the file pointer.
24	Pop a file pointer from the stack and close the corresponding file.

Table 2: Traps in the current SSM.

2 Data Types

The language treated in the lecture notes only has basic data types and no assignments. When a value of basic type is used as function argument, the value is simply pushed to the stack. The situation in SPL is more complicated.

First, there are data types such as tuples and lists. Values of these types are typically built on the heap. A pointer to the data structure is then passed to functions.

Second, there are assignments which can modify objects in place. This means that when handling a variable of such type, there is a difference between the pointer and the object it points to.

3 Calling Semantics

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

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

For data structures, there are more possibilities. You should choose one and stick to it.

- In C and C++, values are stored directly in variables and function arguments. When passing a value to a function, the value is copied. If a value contains pointers to other values, like the tail of a list, only the pointer is copied, not the referenced value. This is known as a *shallow copy*.

  If you choose this variant, you can make copies of objects on the stack. The type information tells you what objects can be expected, so you know how much space to reserve. There is a slight complication with lists. The type system does not tell you whether a list is empty. 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.

- C and C++ also have the notion of pointers. Both variables and function arguments can hold pointers. Assigning to pointer variables only shuffles pointers around, any referenced objects are unaffected. Modifying a referenced object is possible by explicitly dereferencing a pointer. Passing pointers to functions copies only the pointers, not the referenced objects. When a function modifies an object using a pointer, this change is visible to any code that also holds a pointer to the object.

  We do not recommend having both ordinary variables and pointer variables in SPL. However, interpreting every variable and function argument as a pointer is an option.

- Additionally, in C++ there are *by-reference* variables and function arguments. They are marked with a `&`. Such variables behave like pointer arguments with implicit dereferencing.

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

- Java uses *by-value* parameter passing for basic values like integers and Booleans. Objects however are passed *by-reference*, which means pointers to objects get passed to functions, and there is implicit dereferencing.

- Functional languages like Clean always pass pointers to data structures and use implicit dereferencing. Since there are no assignments this is less critical, and it provides a convenient implementation of lazy evaluation.

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

We have deliberately been somewhat vague about the semantics of SPL. In order to implement the code generator, you need to clearly define the behaviour of variables, assignments and parameter passing. You are free to choose the semantics of SPL, but we want you to give a clear description of it. It is not required that this is a complete formal semantics, a concise informal or semiformal description suffices. Furthermore, you must add some well documented test cases showing that your compiler implements the described semantics. This has to be done in the final report.

4 Polymorphism

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

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

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

Another polymorphic function is `reverse :: [t] → [t]`. When the compiler generates code for such a function, the type of the actual argument is not known. The compiler must use an argument handling scheme that works for every type of argument.

5 Overloading

SPL has overloaded operators like == and +. Similar to polymorphism, many types of arguments can be used for such an operator. In contrast to polymorphism, the implementation of the operator depends on the actual types of its arguments. For example, integers are compared directly, while objects on the heap need to be dereferenced first, so that their values are compared, not the pointers. This requires several implementations of == and selecting the right one based on the type of the arguments. Selecting an implementation is called *resolving the overloading*.

In many situations the overloading can be resolved at compile time. For instance, in the expression `x + 7 == y` the integer value `7` indicates that the code generator has to select integer addition. This in turn implies that we need equality for integers and that `y` must have type **Int**.

In overloaded functions like `equal` below, it is not possible to resolve the overloading at compile time.

```
equal ( x, y ) :: t t → Bool {
    return x == y;
}
```

The easiest solution is to forbid this kind of situation.

A more sophisticated solution is to allow this when the overloading can be resolved by the application of the overloaded function, for example:

```
correct ( a ) :: Int → Bool {
    return equal (a, 42);
}
```

Allowing this kind of overloading can require a different version of the overloaded function occurring in the program compiled.

If you want to allow overloading and cannot resolve overloading at compile time, you have to resolve the overloading at run time. Even when a compiler is able to solve the overloading at run time, it will most likely try to resolve it at compile time in order to speedup the generated program.

5.1 Equality

There are two notions of equality commonly found in programming languages. There is *identity* and *equality*. Two objects are identical if they actually are the same object in memory. Two objects are equal if they represent the same value, even if they are at different places in memory.

For example, Java has the equality operator (==) and the equals method. The equality operator checks for pointer equality, and the equals method usually recurses into the data structure to compare.

Implementing true equality for datatypes like lists and tuples is tricky, because it must be parameterized with the equality check for the element types, which in turn might need to be parameterized, to an arbitrary nesting depth.

For SPL it is okay to have (==) be the identity check. If you want, you can let the type checker generate code for nested comparison based on the type structure of the arguments.

5.2 Printing

Printing is similarly complicated to equality. The print functions for lists and tuples need to be parameterized with the print functions of the element types, which in turn might need to be parameterized, to an arbitrary nesting depth. For SPL you can choose between the following, in increasing order of difficulty.

1. Forbid printing of lists and tuples.
2. Print only the outermost layer of nested lists and tuples, and print the elements as pointers.
3. Generate code for nested printing based on the type structure.

6 Memory Management

The creation and deletion of objects is called memory management. Again there are interesting choices to be made. In a simple variant, the deletion of objects is left to the

user, like in C. A more advanced solution uses automatic garbage collection like in Java or functional programming languages.

It is fine to use an even simpler memory management in the assignment. New objects are created as long as there is free memory. When the heap is full the program stops with an appropriate message.

When allocating memory for an object, the compiler needs to know how big the object is going to be. In many cases the type of an object determines its size. Again, lists are a bit more complicated because non-empty lists may need more space than empty lists, depending on your implementation. A safe strategy is to always reserve space for a non-empty list, then the empty list also fits. To distinguish empty lists from non-empty lists at run time, you can use a *tag*, which gives information about the content of a memory cell. A tag is put either in an additional piece of memory or in some situations it fits in unused bits of a memory cell's content. For instance pointers are supposed to be valid heap addresses, but often not every byte of the heap can be addressed individually. When the adressable word size is 4 bytes, then every address is a multiple of four, meaning the lowest two bits of pointers are unused. Another possibility is that when the heap has less memory cells than numbers representable by an integer, some of the highetst bits of pointers are unused.

7 Deliverables

As with the previous assignments, you have to give a presentation about your code generator. The date has been announced on Blackboard and in the lecture. Here are a couple ideas what you can talk about.

- The chosen semantics of SPL, like call-by-reference or call-by-value.
- The compilation schemes used in your compiler. Typical things to explain here are calling conventions, stack management, stack layout, heap layout, and heap management.
- The implementation of lists and tuples.
- The implementation of polymorphic functions.
- The implementation of overloaded functions in the standard library like (==) and **print**.