# Phase 3: Code Generation

Compiler Construction
Radboud University

## 1 Background

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

## 2 Learning Outcome

Implement a code generator for an imperative language. If the specification is incomplete, make and document the well-founded design decisions about the changes.

## 3 Instructions

Implement the code generator for your compiler. The target language of the code generator is the Simple Stack Machine (SSM) language.

## 4 Deliverables

**Presentation** A third of the groups will present this assignment. Which groups have to present this assignment is announced on Brightspace.

In the presentation of this phase you tell the other students how your semantic analyses are set up and you demonstrate it. We all know the language and the theory, so you should just mention things specific to your code generation, like how you handled some interesting language constructions, and other interesting points.

**Report sections** Furthermore, finish the up to and including the fourth chapter of the report[1]. The provided skeleton contains example questions that you can answer but make sure the report is self contained.

## 5 Discussion

During the course you make your own implementation of a complete SPL compiler accompanied by a report that will be tried on the oral exam. The code generation phase on its own is not

[1]The provided skeleton for the report is mandatory and can be found here: `https://gitlab.science.ru.nl/compilerconstruction/material`

graded but only as part of the total project. Comments are provided on the report sections that were handed in for guidance and affirmation.

Criteria for this phase can be found in Table 1.

## 6 SSM

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://gitlab.science.ru.nl/compilerconstruction/ssm`. Feel free to submit patches if you found some way of improving the interpreter.

Detailed descriptions of the available machine instructions can be found in the program's help or here: `https://gitlab.science.ru.nl/compilerconstruction/ssm/-/blob/master/HELP.md`

### 6.1 Heap

SPL has only one level of functions and hence no complications with nested functions. The problems with finding arguments on the stack does not 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.

See the instructionset documentation for information about the available instructions to manipulate the heap.

The creation and deletion of objects is called memory management. It is fine to use the simplest memory management possible 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 and you can use a tag to distinguish the two. Another approach may be to use a `NULL` pointer to represent the empty list.

## 7 Data Types

In SPL there are several classes of data types.

First, the basic types, integers, booleans and characters, can just be store as is and copied when needed elsewhere.

Second, 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.

Third, 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.

## 8 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 CPP, 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 CPP 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 CPP 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.

The semantics description of SPL is deliberately kept somewhat vague. 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.

## 9 Polymorphism

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

```
id (x : t) : t {
    return x;
}
k (x : a, y : b) : a {
    return x;
}
```

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

## 10 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 the compile time of the function.

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

The easiest solution is to forbid this kind of situation.

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

```
correct (x : Int, y : Int) : Bool {
    return equal (x, y);
}
```

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

### 10.1 Equality (and printing)

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 all datatypes like lists and tuples is tricky, because it either must be parameterized with the equality check for the element types or the complete equality function for the compound type must be generated to an arbitrary nesting depth.

| Property | Insufficient | Sufficient | Good | Optional |
|---|---|---|---|---|
| Design | No motivated design decisions. | Motivated design decisions. | | |
| Errors | No error handling. | Errors that refer to an AST node. | Localised errors that are as precise as possible. | |
| Data representation | Lists cannot be nested. | Lists can be nested. | Lists are stored efficiently. | |
| Overloading and polymorphism | Polymorph functions are monomorphised. | Overloaded builtins are monomorphised and implemented for basic types. | Overloaded builtins are monomorphised implemented for all types. | Overloaded builtins may be called to polymorph function (it makes them overloaded). |
| General | Non-working code. | Working code with a good I/O interface. | Well structured modular code. | Use of CI and issue tracker, other useful extensions regarding this phase. |

Table 1: Assignment 3 rubric