Compiler Construction part 3: Code generation

Joost Rijneveld, s4048911
Koen van Ingen, s4058038

1 Overview

Usage: `python src/toolchain.py spl/assignmentprogs.spl`

Our submission contains the following files:

src/toolchain.py The toolchain calls the scanner, parser, printer and type checker consecutively, passing the result from one on to the next. This is the script you need to execute to parse, print and type check an `.spl` source file.

src/scanner.py This file contains the scanner. The scanner expects a filename and produces a list of Tokens that can then be passed on to the parser. `Token` objects contain a position (line / col) to indicate where the token was found, for future error handling.

src/parser.py The parser expects a list of `Token` objects, as produced by the scanner. It then outputs the root of a parse tree (a `Node` object). Grammar rules can be mapped one-to-one to `parse_` functions, some of which are generated based on a blueprint function that is partially applied (the `tail_recursion` function). When the parser runs into an error, it prints the expected literal (if applicable) and the line and column number.

src/semanticanalysis.py This is the type and function binding checker. It will check for type errors and function binding errors, when it is called with the `check_binding` function. It expects a tree that can be generated with `parser.py`. A dictionary with a symbol table can be given as an optional argument. The `Symbol` class is also defined in this file. We currently use this optional argument to give the function a symbol table with the two predefined functions: `isEmpty` and `print`.

src/printer.py This script expects a parse tree as provided by the parser, and prints formatted (and aligned) SPL source code to stdout.

spl/* This folder contains the 10 sample SPL programs, as required by the assignment. All of them parse, print and type check fine. Some of them have wrong statements (with type errors) in comments, which can be enabled to trigger type check errors. In section 7, we explain these programs and their result in more details.

grammar.txt This file contains the modified grammar which we used for our compiler.

2 Used grammar

```
Decl    = VarDecl
        | FunDecl

VarDecl = Type id "=" Exp ";"

FunDecl = "Void" id "(" [ Fargs ] ")" "{" VarDecl* Stmt+ "}"
        | Type id "(" [ Fargs ] ")" "{" VarDecl* Stmt+ "}"

Type    = "Int"
        | "Bool"
        | "(" Type "," Type ")"
        | "[" Type "]"
        | id

Fargs   = Type id [ "," Fargs ]

Stmt    = "{" Stmt* "}"
        | "if" "(" Exp ")" Stmt [ "else" Stmt ]
        | "while" "(" Exp ")" Stmt
        | ExpFunc ";"
        | ExpField "=" Exp ";"
        | "return" [ Exp ] ";"

Exp     = ExpOr
ExpOr   = ExpAnd    ("||" ExpAnd)*
ExpAnd  = ExpEq     ("&&" ExpEq)*
ExpEq   = ExpCmp    (("==" | "!=") ExpCmp)*
ExpCmp  = ExpAdd    (("<" | "<=" | ">" | ">=") ExpAdd)*
ExpAdd  = ExpMult   (("+" | "-") ExpMult)*
ExpMult = ExpCon    (("*" | "/" | "%") ExpCon)*
ExpCon  = ExpUn     ":" ExpCon
        | ExpUn
ExpUn   = ("-" | "!") ExpUn
        | ExpBase
ExpBase = ExpField
        | ExpFunc
        | int
        | bool
        | "[" "]"
        | "(" Exp ")"
        | "(" Exp "," Exp ")"

ExpField = id (".hd" | ".tl" | ".fst" | ".snd")*
ExpFunc  = id "(" [ ExpArgs ] ")"
ExpArgs  = Exp [ "," ExpArgs ]

int     = digit+
id      = alpha ("_" | alphaNum)*
bool    = True | False
```

3 Typing rules

Note that many of these rules are directly derived from the rules as provided for SL in the lecture slides, or are slight variations thereof.

The two basic types, `Int` and `Bool`, are defined by the following axioms. Note that integers are taken from $\mathbb{N}$ rather than $\mathbb{Z}$, as negative integers are typed using the unary negation operator.

$$\frac{b \in \{\texttt{True},\ \texttt{False}\}}{\Gamma \vdash b : \texttt{Bool}}\ (\textsc{Bool})$$

$$\frac{i \in \mathbb{N}}{\Gamma \vdash i : \texttt{Int}}\ (\textsc{Int})$$

To deal with the operators, we rely on the following two rules – one for binary and one of unary operators, respectively.

$$\frac{\odot : \sigma_1 \to \sigma_2 \to \tau \qquad \Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash e_1 \odot e_2 : \tau}\ (\textsc{BinOp})$$

$$\frac{\odot : \sigma \to \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash \odot e : \tau}\ (\textsc{UnOp})$$

These rules can be 'instantiated' by applying the axioms below.

$$\frac{}{\odot \in \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%}\} : \texttt{Int} \to \texttt{Int} \to \texttt{Int}}\ (\textsc{Arith})$$

$$\frac{}{\odot \in \{\texttt{<=},\ \texttt{<},\ \texttt{>=},\ \texttt{>}\} : \texttt{Int} \to \texttt{Int} \to \texttt{Bool}}\ (\textsc{Cmp})$$

$$\frac{}{\odot \in \{\texttt{==},\ \texttt{!=}\} : \sigma \to \sigma \to \texttt{Bool}}\ (\textsc{GenCmp})$$

$$\frac{}{\odot \in \{\texttt{\&\&},\ \texttt{||}\} : \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}}\ (\textsc{BoolOp})$$

$$\frac{}{\texttt{:} : \sigma \to [\sigma] \to [\sigma]}\ (\textsc{Cons})$$

$$\frac{}{\texttt{-} : \texttt{Int} \to \texttt{Int}}\ (\textsc{Neg})$$

$$\frac{}{\texttt{-} : \texttt{Bool} \to \texttt{Bool}}\ (\textsc{Inv})$$

Note that the 'cons'-operator is often dealt with separately, but this is not required for our specific case. Complications regarding associativity are already dealt with in the grammar. This means we can just include it as a binary operator, above. Creating tuples does require a distinct rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2) : (\sigma_1, \sigma_2)}\ (\textsc{Tuple})$$

Variables can be derived from the environment as follows:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash \texttt{x} : \sigma}\ (\textsc{Var})$$

For function calls, instances of the following type rule can be used:

$$\frac{\texttt{f} : \sigma_1 \to \ldots \to \sigma_n \to \tau \in \Gamma}{\Gamma \vdash \texttt{f(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} : \tau}\ (\textsc{Fn})$$

We can now create axioms for the `isEmpty` and `print`-functions using instances of the above rule:

$$\frac{}{\texttt{isEmpty} : [\sigma] \to \texttt{Bool} \in \Gamma}\ (\textsc{Fn}) \qquad \frac{}{\texttt{print} : \sigma \to \texttt{Void} \in \Gamma}\ (\textsc{Fn})$$

The functions `hd`, `tl`, `fst` and `snd` are written postfix in SPL, so it requires a transformation to get them to fit to the above rules. We define axioms to add their transformed forms to the environment, and unary transformation rules such that they fit in (Fn). Note that this could be resolved in a more compact fashion by immediately jumping to the axioms, but the approach below provides more consistency with 'regular' functions.

$$\frac{}{\texttt{hd} : [\sigma] \to \sigma \in \Gamma}\ (\textsc{Hd}) \qquad \frac{}{\texttt{fst} : (\sigma_1, \sigma_2) \to \sigma_1 \in \Gamma}\ (\textsc{Fst})$$

$$\frac{}{\texttt{tl} : [\sigma] \to [\sigma] \in \Gamma}\ (\textsc{Tl}) \qquad \frac{}{\texttt{snd} : (\sigma_1, \sigma_2) \to \sigma_2 \in \Gamma}\ (\textsc{Snd})$$

Applying the functions is then done using the following rules:

$$\frac{\Gamma \vdash \texttt{hd}(e) : \sigma}{\Gamma \vdash e.\texttt{hd} : \sigma}\ (\textsc{HdApp}) \qquad \frac{\Gamma \vdash \texttt{fst}(e) : \sigma}{\Gamma \vdash e.\texttt{fst} : \sigma}\ (\textsc{FstApp})$$

$$\frac{\Gamma \vdash \texttt{tl}(e) : \sigma}{\Gamma \vdash e.\texttt{tl} : \sigma}\ (\textsc{TlApp}) \qquad \frac{\Gamma \vdash \texttt{snd}(e) : \sigma}{\Gamma \vdash e.\texttt{snd} : \sigma}\ (\textsc{SndApp})$$

Custom functions can be added using definitions, and applied in an identical fashion.

4 Scoping

As we construct a global symbol table and function-specific symbol tables that inherit from the global table, a lot of the scoping hurdles have already been covered. In terms of environments, we first create a global environment Γ, and then create a duplicate Γ' when checking a function. Variables that are defined within a function are added to Γ', leaving Γ untouched. Arguments are treated identically; they are functionally equivalent to variable declarations. Note that as there is no notion of functions within functions, the nesting depth of environments is limited to two.

When variables are defined in Γ' that already exist in the global scope Γ, they shadow the globally defined variable. This produces a warning, but does not lead to a fatal exception. However, variables should never be defined twice within the same environment; attempting to add a variable to some environment Γ when it is already included leads to a fatal exception. This also includes overriding function arguments.

It should also be noted that definitions of functions and variables can be in any order, as they are first added to the respective environments before the actual expressions are validated. They do need to be declared at one point: implicit declaration through assignment is not allowed. This also implies that there is no need to take care of `if`-scopes, as these cannot contain new variable declarations. It is, however, important to note that the declaration order of variables is important for binding analysis (i.e. `x = y; y = x;` is not allowed).

With respect to generic types, it should perhaps be noted that generic types specified in the function header are bound by argument types, and cannot be used outside the function. This implies that duplicate generic type names between function definitions and function calls do not conflict.

All scoping rules are explicitly enforced through fatal errors, meaning that the usage of unbound variables or the calling undefined functions does not result in undefined behaviour (like in C), but simply terminates immediately with an exception.

5 Semantics of SPL

The syntax of SPL was already obvious from the start of our compiler, because a grammar of SPL was provided. This was not the case with the semantics and thus we needed to make decisions about the semantics from the very beginning. One of the first decisions we made was about the order of operators: we needed to define the operator associativity already in the parser. Here we chose to use the same approach as C, as found on `http://en.wikipedia.org/wiki/C_operators`.

SPL has support for the primitive data types like Integers, Booleans as well as more complex data types like lists and tuples. The primitive data types are stored on the stack, and thus are always called by value when we use them as a function argument. Lists and tuples are stored on the heap, with only a pointer on the stack remaining. Therefore, this data types are called by reference when used as function arguments. This also has some other implications: an equality on lists and tuples will just compare the pointers instead of the whole lists. Another implication is the assignment of a part (ie the tail) of a list, which can be shown with an example:

```
[Int] x = 1 : 2 : 3 : [];
[Int] y = 5 : 6 : 7 : [];
x.tl = y; // Just a pointer will be changed here
y.hd = 1; // This will also change x
```

In this example will the tail of the list x be pointed to y. The list will contain then the following elements: $[1, 5, 6, 7, [\,]]$. If we change the head of y to 1, then the second element of x will also be changed to 1, resulting in the following elements: $[1, 1, 6, 7, [\,]]$.

Beside the decisions about pointers, we also needed to determine how a SPL program can be started. The grammer does not obligate you to construct a main function. We decided to keep it this way. Therefore, you need to create at least one variable that calls a function (for example a self constructed main function) to let something interesting happen. You can for example make a 'Start' variable:

```
Int Start = main();

Int main() {
        return 1;
}
```

6 Compilation Schemes

6.1 Memory Management: The stack and the heap

SSM contains a heap and a stack that can be used for memory management. The stack has far more instructions for data manipulation than the heap, which is one of the reasons to use it as a dynamic working space. The heap is used for the static data, like the data inside tuples and lists. We also use the heap for global variables. All the other relevant data is stored on the stack. We will explain this in more detail and illustrate it with some examples.

6.1.1 Global variables

Global variables are stored on the heap. Our compiler will first reserve space for all the global variables when it compiles an SPL program. After this space has been reserved, we initialise the value of the variables, by recursively generating the expressions (the right side of a variable assignment is an expression in SPL). An example:

The SPL:

```
Int x = 5;
Bool y = True;
```

Will translate to:

```
ldc 2002     ;   Reserve 2002-2000 = 2 space for 2 global vars
str HP       ;   More HP +2
ldc 5        ;   Load value for x (parse expression for x)
ldc 2000     ;   Load address of x
sta 0        ;   Store value (=5) at address of x in heap
ldc -1       ;   Load value for y (bool True is encoded as -1)
ldc 2001     ;   Load address of y
sta 0        ;   Store value (=True=-1) at address of y in heap
halt
```

6.1.2 Function Calls

When we call a function, we will need to copy the arguments of the function call to the stack space of this function. By it this way, the function arguments will be passed by value. Because a function needs space for its return address and the mark pointer (which will be explained later), we need to place the arguments at 2 stack positions further than our current stack pointer. Therefore, we need to increment the stack pointer, place the arguments and set the stack pointer back to the original position. When we then call the function, the functions arguments will already be at the right place. An example:

```
Int x = id(5);
Int id(Int y)
{
    Int z = 5;
    return y;
}
```

The function call of this SPL program will look like:

```
ldc 2002     ;   Reserve 2001-2000 = 1 space for 1 global var
str HP       ;   More HP +1
ajs 2        ;   Move SP to set function argument
ldc 5        ;   Load function argument
ajs -3       ;   Move SP back
bsr id       ;   Call function
ldr RR       ;   Get return value (=5)
ldc 2000     ;   Get address of x
sta 0        ;   Store return value (=5) at address of x in heap
halt
```

6.1.3 Function declarations

Inside a function, we need to reserve space for both the function arguments as well as the local variables. SPL has a convenient instruction for this purpose, namely the `link n` instruction. This instruction moves the stack pointer n positions (where n is the amount of variables you have). After this reservation, it will place the mark pointer at the top of the stack. The mark pointer can then be used to refer to the local variables. We will explain this with an example:

When we recall the id function of our last SPL program, then the function declaration will in SSM look like:

```
id:         ;   Just a label, used to branch to this address
link 2      ;   Reserve space for 2 vars: argument y and local var z
ldc 5       ;   Load value for y
stl 2       ;   Store in second local var (2 is address of mark pointer +2)
ldl 1       ;   Load the first local var (=x and already copied to this
            ;   position in the function call)
str RR      ;   Store x in the return register
unlink      ;   Clean the reserved space up (local vars are no longer
            ;   needed after the function)
ret         ;   Return to original address of function call
```

6.1.4 Lists and tuples

Lists and tuples are stored in the heap. On the stack we maintain a pointer to the first element in the heap if it is a local variable. For global lists and tuples, we maintain this first pointer on the heap as well (at the same location as the other global variables). When we call a function with a list/tuple as argument, we give this function the pointer to this list/tuple. Therefore lists and tuples are always passed by reference instead of by value.

We first show an example of a list of the integers 1,2,3. We need to store every element plus a pointer to next element on the heap. The pointer of the last element is a null-pointer, to denote the end of the list:

```
[Int] x = 1: 2: 3: [];
```

```
ldc 2001    ;   Reserve 2001-2000 = 1 space for 1 global var
str HP      ;   More HP +1
ldc 1       ;   Load first element of the list
ldc 2       ;   Load second element of the list
ldc 3       ;   Load third element of the list
ldc 0       ;   Load null-pointer for empty list
sth         ;   Store this null-pointer on the heap (returns pointer stack)
ajs -1      ;   Throw returned heap address away
sth         ;   Store third element on the heap (return pointer to stack)
sth         ;   Store pointer to third element on the heap
ajs -1      ;   Throw returned heap address away
sth         ;   Store second element on the heap (return pointer to stack)
sth         ;   Store pointer to second element on the heap
ajs -1      ;   Throw returned heap address away
sth         ;   Store first element on the heap (return pointer to stack)
ldc 2000    ;   Load heap address of global var x
sta 0       ;   Store pointer to first element in global var x
halt
```

After execution of this program, the heap will look like:

Address	Value	Description
000007d0	000007d6	Ptr of x to list
000007d1	00000000	Null-ptr (last element)
000007d2	00000003	Third element
000007d3	000007d2	Ptr to third element
000007d4	00000002	Second element
000007d5	000007d4	Ptr to second element
000007d6	00000001	First element

Tuples are also stored in the heap, but there is no need to combine each element with a pointer to the next element, since the length of tuples is constant and known at compile-time. An example:

```
(Int,Int) x = (5, 6);
```

```
ldc 2001    ;   Reserve 2001-2000 = 1 space for 1 global var
str HP      ;   More HP +1
ldc 5       ;   Load first element of the tuple
ldc 6       ;   Load second element of the tuple
sth         ;   Store the second element on the heap
ajs -1      ;   Throw returned heap address away
sth         ;   Store the first element on the heap
ldc 2000    ;   Load heap address of global var x
sta 0       ;   Store pointer to first element in global var x
halt
```

After execution of this program, the heap will look like:

Address	Value	Description
000007d0	000007d2	Ptr of x to tuple
000007d1	00000006	Second element
000007d2	00000005	First element

We can also combine lists and tuples, as shown in the next example:

```
([Int], Int) x = ( 1 : 2 : [], 6);
```

Which will result in the following SPL:

```
ldc 2001    ;   Reserve 2001-2000 = 1 space for 1 global var
str HP      ;   More HP +1
ldc 1       ;   Load first element of the list
ldc 2       ;   Load second element of the list
ldc 0       ;   Load null-pointer for empty list
sth         ;   Store the null-pointer on the heap
ajs -1      ;   Throw returned heap address away
sth         ;   Store the second element on the heap
sth         ;   Store second element on the heap (return pointer to stack)
ajs -1      ;   Throw returned heap address away
sth         ;   Store third element on the heap (return pointer to stack)
ldc 6       ;   Load second element of tuple (first value is ptr to list)
sth         ;   Store second element on the heap
ajs -1      ;   Throw returned heap address away
sth         ;   Store first element on the heap (is a ptr to list)
ldc 2000    ;   Load heap address of global var x
sta 0       ;   Store pointer to first element in global var x
halt
```

After execution, the heap will look like:

Address	Value	Description
000007d0	000007d6	Ptr of x to tuple
000007d1	00000000	Null-ptr (last element)
000007d2	00000002	Second element of list
000007d3	000007d2	Ptr to second element of list
000007d4	00000001	First element of list
000007d5	00000006	Second element of tuple
000007d6	000007d4	First element of tuple (ptr to list)

6.2 Branching

SPL has support for both `if-else` and `while`-statements. In an `if`-statement, to determine whether we need to execute the if-part or the else-part, we need to check the condition. After this check we need to branch over the `if` or the `else`, and thus we need to count the number of instructions that needs to be skipped. While SSM also has support for jumping to labels, we decided not to use it because it was more convenient not to have to worry about making unique labels in the case of deep nested `if`-statements. An example of an `if-else`-construction:

```
Int x = main();

Int main() {
    if ( 3 > 4 )
        return 5;
    else
        return 6;
}
```

Which will result in the following SSM:

```
ldc 2001     ;   Reserve 2001-2000 = 1 space for 1 global var
str HP       ;   More HP +1
bsr main     ;   Call main function
ldr RR       ;   Load return value (=6)
ldc 2000     ;   Load heap address of global var x
sta 0        ;   Store return value in global x
halt
main:
    link 0   ;   Reserve space for locals (0 in this case)
    ldc 3    ;   Load first constant for expression
    ldc 4    ;   Load second constant for expression
    gt       ;   Compare two values, place true/false on stack
    brf 8    ;   Branch if false 8 instructions (to else statement)
    ldc 5    ;   Load return value in if-statement
    str RR   ;   Store x in the return register
    unlink   ;   Clean the reserved space up
    ret      ;   Return to original address of function call
    bra 6    ;   Skip the else statement
    ldc 6    ;   Load return value in else-statement
    str RR   ;   Store x in the return register
    unlink   ;   Clean the reserved space up
    ret      ;   Return to original address of function call
```

The `brf 8` instruction jumps over the `else` statement. We need to jump 8 addresses, because the `if`-statement uses this space: 2 instructions without arguments + 3 instructions with arguments, where the instructions with arguments use 2 addresses. This results in a total of 8 addresses.

6.3 Builtin functions

SPL has support for two built-in functions, namely the `print` and the `isEmpty` function. We decided to hardcode these functions into the assembly. They look like this:

```
print:
link 1       ;   Reserve space for 1 local (print takes 1 argument)
ldl 1        ;   Load the first local argument onto the stack
trap 0       ;   Print top topmost element of the stack
```

```
5  unlink      ;   Clean the reserved space up
6  ret         ;   Return to original address of function call
```

```
1  isEmpty:
2  link 1      ;   Reserve space for 1 local (print takes 1 argument)
3  ldl 1       ;   Load the first local argument onto the stack
4  ldc 0       ;   Load constant used for comparison (= null-ptr)
5  eq          ;   Compare list element with 0
6  str RR      ;   Store result of comparison in return register
7  unlink      ;   Clean the reserved space up
8  ret         ;   Return to original address of function call
```

7 Example tests

7.1 General function application

The file `fnapplication.spl` is an example of a program that tests the function application and whether the number and type of arguments of the function call matches with the header.

```
1  Void foo(Int x) {
2      [Int] xs = x : [];
3      return;
4      // x = x + 1; // This is dead code
5  }
6
7  Void call_foo() {
8      foo(5);       // This should go right
9      //foo();      // This should go wrong
10     //foo(5, 6); // This should go wrong
11 }
```

The call on line 8 is the only correct function call in this program: only in this case do the number and type of arguments match. In the other cases, we get an error:

```
Exception: [Line 8:2] Incompatible number of arguments for function 'foo'.
  Arguments expected: 1
    Arguments found: 0
```

Another error is the unreachable code on line 4: because this code is listed after a return statement, this cannot be executed. Our compiler does not accept this code and produces an error:

```
Exception: [Line 4:4] Unreachable code detected
  If this is intentional, enclose it in comments
```

If we fix both errors, we get the following pretty-printed program and symbol table:

```
Void foo(Int x)
{
        [Int] xs = (x : []);
        return;
}
Void call_foo()
{
        foo(5);
}

=====================================================================
Position     Name            Type            Argtypes
---------------------------------------------------------------------
0:0          print           'Void'          't'
1:6          foo             'Void'          'Int'
0:0          isEmpty         'Bool'          ['t']
6:6          call_foo        'Void'
=====================================================================
Position     Name            Type            Argtypes
---------------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
1:14         x               'Int'           None
0:0          print           'Void'          't'
```

```
2:8          xs              ['Int']         None
1:6          foo             'Void'          'Int'
6:6          call_foo        'Void'
================================================================
Position     Name            Type            Argtypes
----------------------------------------------------------------
0:0          print           'Void'          't'
1:6          foo             'Void'          'Int'
0:0          isEmpty         'Bool'          ['t']
6:6          call_foo        'Void'
================================================================
```

In this output, the first table refers to the global symbol table and the second to the function `foo()` (which also includes all the global symbols). The third table refers to the function `call_foo()`, but does not contain any arguments or variable declarations.

7.2 Assignments with empty lists

Because empty lists can cause a lot of trouble with type checking (since they can match on any list types), we thought it might be a good idea to include some test variable declarations with empty lists as an example program. This can be found in `emptylists.spl`.

```
1 [[Int]] a = [] : (5 : []) : []; // This should go right
2 //[Int] b =  [] : [];          // This should go wrong
3 [[Int]] c =  [] : [];          // This should go right
4 [[a]] d =  [] : [];            // This should go right
5 //[[a]] e =  (5 : []) : [];    // This should go wrong because a != Int
6
7 // A hard one, should go right:
8 ([Bool], [Int])] f = ([],5:[]) : (True:[],[]) : [];
```

The first one should obviously go right, because every element in the list can be map to a list of integers, which will result in a list of lists of integers. For the second one, this is not the case: we expect a list of integers, but a list with 2 empty lists will be mapped to a list of lists of items, and thus we get an error here:

```
Exception: [Line 2:7] Invalid assignment for id b
  Expected expression of type: ['Int']
    But got value of type: [[None]]
```

The examples on line 4 and 5 should go right, since in both cases, the empty list can be mapped to either a list of lists of Int or a generic list of lists of 'a'. Example 5 will fail because the integer 5 cannot be mapped to the generic type 'a'. Example 5 is actually the same example as the one we show in section 7.4. The example on line 8 combines the other examples with tuples. This should go right if all the other cases are type checked correctly. This example is typed correctly with our type checker as well.

This is the pretty-printed output and the symbtol table of this program:

```
[[Int]] a = ([] : ((5 : []) : []));
[[Int]] c = ([] : []);
[[a]] d = ([] : []);
[([Bool],[Int])] f = (([], (5 : [])) : (((True : []), []) : []));

========================== _glob ============================
Position     Name            Type            Argtypes
----------------------------------------------------------------
```

```
1:9          a               [['Int']]       None
3:9          c               [['Int']]       None
0:0          isEmpty         'Bool'          ['t']
4:7          d               [['a']]         None
8:19         f               [(['Bool'], ['Int'])] None
0:0          print           'Void'          't'
============================================================
```

7.3 Function application with empty lists

In our next example, we combine our function application with the empty lists. The code can be found in `fnonemptylists.spl`.

```
1  [b] foo([b] x) {
2      return x;
3  }
4
5  a bar([a] x) {
6      return x.hd;
7  }
8
9
10 [Int] a = foo(5 : []); // This should go right
11 [Int] b = foo([]);     // This will go right, because [None] fits in [Int]
12 Int c = bar([]);       // Similarly, this should work too
```

The call on line 5 works, because we give this function a list of integers, which is possible because this can be mapped to the more generic type 'b'. As a result, we get back a list of integers, which fits exactly in our variable 'a' that is of type [Int]. For the call on line 6, the same holds, since the empty list fits in a list of integers.

This is the pretty-printed output and symbol table of this program:

```
[b] foo([b] x)
{
  return x;
}
a bar([a] x)
{
  return x.hd;
}
[Int] a = foo((5 : []));
[Int] b = foo([]);
Int c = bar([]);

============================ _glob ============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
10:7         a               ['Int']         None
12:5         c               'Int'           None
11:7         b               ['Int']         None
5:3          bar             'a'             ['a']
0:0          print           'Void'          't'
0:0          isEmpty         'Bool'          ['t']
1:5          foo             ['b']           ['b']
```

```
============================================================
============================ foo ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          print           'Void'          't'
0:0          isEmpty         'Bool'          ['t']
1:5          foo             ['b']           ['b']
5:3          bar             'a'             ['a']
1:13         x               ['b']           None
============================================================
============================ bar ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
10:7         a               ['Int']         None
11:7         b               ['Int']         None
5:3          bar             'a'             ['a']
5:11         x               ['a']           None
0:0          print           'Void'          't'
0:0          isEmpty         'Bool'          ['t']
1:5          foo             ['b']           ['b']
============================================================
```

7.4 Variable assignments

It is not allowed to assign an integer to a generic type, since these types do not match. Our next example demonstrates this. The code can be found in `assignments.spl`.

```
1  Int inc(Int x) {
2      return x + 1;
3  }
4
5  Int count_to_ten() {
6      Int x = 0;
7      while (x <= 10) {
8          x = inc(x);
9      }
10     return x;
11 }
12
13 a identity(a x) {
14     return x;
15 }
16
17 Void main() {
18     // Bool x = y; // This should break: y is not defined
19     Bool y = identity(True);
20     // a x = 0; // This should break: 'a' != 'Int'
21     //inc(x);
22     return;
23 }
```

The assignment on line 18 should go wrong, because 'y' is unknown here and thus cannot be assigned to 'x':

```
Exception: [Line 18:11] Found id y, but it has not been defined
```

The assignment on line 20 should also go wrong, since type ‘a’ is not equal to type ‘Int’. And indeed, we get an error when we uncomment that line:

```
Exception: [Line 20:4] Invalid assignment for id x
  Expected expression of type: 'a'
  But got value of type: 'Int'
```

Otherwise, we get the following pretty-printed code and symbol table:

```
Int inc(Int x)
{
        return (x + 1);
}
Int count_to_ten()
{
        Int x = 0;
        while ((x <= 10))
        {
                x = inc(x);
        }
        return x;
}
a identity(a x)
{
        return x;
}
Void main()
{
        Bool y = identity(True);
        return;
}

======================== count_to_ten ========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
5:5          count_to_ten    'Int'
0:0          print           'Void'          't'
6:6          x               'Int'           None
17:6         main            'Void'
13:3         identity        'a'             'a'
1:5          inc             'Int'           'Int'
============================================================
============================ main ============================
Position     Name            Type            Argtypes
------------------------------------------------------------
19:7         y               'Bool'          None
0:0          isEmpty         'Bool'          ['t']
5:5          count_to_ten    'Int'
0:0          print           'Void'          't'
17:6         main            'Void'
13:3         identity        'a'             'a'
1:5          inc             'Int'           'Int'
============================================================
========================== _glob ===========================
```

```
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
5:5          count_to_ten    'Int'
0:0          print           'Void'          't'
17:6         main            'Void'
13:3         identity        'a'             'a'
1:5          inc             'Int'           'Int'
============================================================
========================== identity ========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
5:5          count_to_ten    'Int'
0:0          print           'Void'          't'
13:14        x               'a'             None
17:6         main            'Void'
13:3         identity        'a'             'a'
1:5          inc             'Int'           'Int'
============================================================
============================ inc ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
5:5          count_to_ten    'Int'
0:0          print           'Void'          't'
1:13         x               'Int'           None
17:6         main            'Void'
13:3         identity        'a'             'a'
1:5          inc             'Int'           'Int'
============================================================
```

7.5 Assignment in lists

With this example, we want to show that it is possible to overwrite a part of a list with another list. In general this shows that we can do field assignments. In assembly, this will be fixed by changing the pointers. We demonstrate this with the following SPL code:

```
1  Int x = main();
2
3  Int main() {
4      [Int] y = 1 : 2 : 3 : [];
5      [Int] z = 7 : 8 : 9 : [];
6      y.tl.tl = z;
7
8      return y.tl.tl.hd; // Should be 7
9  }
```

This will produce the following symbol table and pretty-printed program:

```
Int x = main();
Int main()
{
```

```
        [Int] y = (1 : (2 : (3 : [])));
        [Int] z = (7 : (8 : (9 : [])));
        y.tl.tl = z;
        return y.tl.tl.hd;
}

============================ _glob ============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
1:5          x               'Int'           None
3:5          main            'Int'
0:0          isEmpty         'Bool'          ['t']
=============================================================
============================ main =============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
4:8          y               ['Int']         None
3:5          main            'Int'
0:0          isEmpty         'Bool'          ['t']
5:8          z               ['Int']         None
=============================================================
```

7.6 Unequal comparison

When we want to sort two lists and compare them, they should be of the same type. The program in the file `unequalcomparisongenerics.spl` gives an example of this problem. This is the program that was briefly discussed on the Blackboard at the second seminar.

```
1  [a] sort([a] list) {
2      return list;
3  }
4
5  Bool main() {
6      [Int] as = 1:2:3:[];
7      [Bool] bs = True:[];
8      //return sort(as) == sort(bs); // Wrong: since Int != Bool
9      return sort(as) == sort([]);
10 }
```

The call on line 8 will produce an error, since a returned list of integers cannot be compared with a returned list of booleans. We will get the following error:

```
Exception: [Line 8:18] Incompatible types for operator '=='
  Types expected: 't', 't'
    Types found: ['Int'], ['Bool']
```

With the empty list on line 9, everything will go right, and we get the following pretty-printed program and symbol table:

```
[a] sort([a] list)
{
        return list;
}
```

```
Bool main()
{
        [Int] as = (1 : (2 : (3 : [])));
        [Bool] bs = (True : []);
        return (sort(as) == sort([]));
}
```

```
============================ _glob ============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
1:5          sort            ['a']           ['a']
5:6          main            'Bool'
0:0          isEmpty         'Bool'          ['t']
=============================================================
============================ main =============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
1:5          sort            ['a']           ['a']
6:8          as              ['Int']         None
0:0          isEmpty         'Bool'          ['t']
7:9          bs              ['Bool']        None
0:0          print           'Void'          't'
5:6          main            'Bool'
=============================================================
============================ sort =============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
1:5          sort            ['a']           ['a']
1:14         list            ['a']           None
5:6          main            'Bool'
0:0          isEmpty         'Bool'          ['t']
=============================================================
```

7.7 Type of operators

Another interesting example is the use of different operators. We show this with a program that calculates the product of the values in a list. This program can be found in `operators.spl`.

```
1  Int product ( [Int] list ) {
2      if (isEmpty(list)) {
3          return 1;
4      }
5      return list.hd * product(list.tl);
6  }
7
8  Bool all([Bool] list) {
9      if (isEmpty(list)) {
10         return True;
11     }
12     return list.hd && all(list.tl);
13 }
14
```

```
15 Bool any([Bool] list) {
16     if (isEmpty(list)) {
17         return True;
18     }
19     return list.hd || any(list.tl);
20 }
```

This is a rather basic program, since no generic types are used at all. However, we can use it to show that the *-operator expects two integers as arguments: if we replace the [Int] in the function header with [a], we get an error:

```
Exception: [Line 5:17] Incompatible types for operator '*'
  Types expected: 'Int', 'Int'
  Types found: 'a', 'Int'
```

The same holds for the *all* and *any* function: when we replace the boolean type with a generic type, we get the same exception:

```
Exception: [Line 12:17] Incompatible types for operator '&&'
  Types expected: 'Bool', 'Bool'
  Types found: 'a', 'Bool'
```

With the right types in the header, we get the following pretty-printed program and symbol table:

```
Int product([Int] list)
{
        if (isEmpty(list))
        {
                return 1;
        }
        return (list.hd * product(list.tl));
}
Bool all([Bool] list)
{
        if (isEmpty(list))
        {
                return True;
        }
        return (list.hd && all(list.tl));
}
Bool any([Bool] list)
{
        if (isEmpty(list))
        {
                return True;
        }
        return (list.hd || any(list.tl));
}
```

```
========================== product ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
1:5          product         'Int'           ['Int']
0:0          isEmpty         'Bool'          ['t']
0:0          print           'Void'          't'
1:21         list            ['Int']         None
```

```
8:6          all             'Bool'          ['Bool']
15:6         any             'Bool'          ['Bool']
============================================================
============================ all ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
1:5          product         'Int'           ['Int']
0:0          isEmpty         'Bool'          ['t']
0:0          print           'Void'          't'
8:17         list            ['Bool']        None
8:6          all             'Bool'          ['Bool']
15:6         any             'Bool'          ['Bool']
============================================================
============================ _glob =========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          print           'Void'          't'
1:5          product         'Int'           ['Int']
0:0          isEmpty         'Bool'          ['t']
8:6          all             'Bool'          ['Bool']
15:6         any             'Bool'          ['Bool']
============================================================
============================ any ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
1:5          product         'Int'           ['Int']
0:0          isEmpty         'Bool'          ['t']
0:0          print           'Void'          't'
15:17        list            ['Bool']        None
8:6          all             'Bool'          ['Bool']
15:6         any             'Bool'          ['Bool']
============================================================
```

7.8 Zip and tuples

From the example of the zip function (see below), it follows that we are able to handle both lists and tuples with generic types. The code of this program can be found in `zip.spl`.

```
1  [(a,b)] zip ([a] a,[b] b) {
2      if ( isEmpty(a) || isEmpty(b) ) {
3          return [];
4      }
5      return (a.hd,b.hd) : zip (a.tl,b.tl);
6  }
7
8  Int main()
9  {
10     [Int] ints = 1 : 2 : 3 :4 : [];
11     [Bool] bools = True : False: False: True : [];
12     [(Int,Bool)] zipped = zip(ints,bools);
13     return 1;
14 }
```

When we run this program, we get the following pretty-printed code and symbol table.

```
[(a,b)] zip([a] a, [b] b)
{
        if ((isEmpty(a) || isEmpty(b)))
        {
                return [];
        }
        return ((a.hd, b.hd) : zip(a.tl, b.tl));
}
Int main()
{
        [Int] ints = (1 : (2 : (3 : (4 : []))));
        [Bool] bools = (True : (False : (False : (True : []))));
        [(Int,Bool)] zipped = zip(ints, bools);
        return 1;
}
```

```
=========================== _glob ============================
Position     Name            Type            Argtypes
--------------------------------------------------------------
0:0          print           'Void'          't'
8:5          main            'Int'
0:0          isEmpty         'Bool'          ['t']
1:9          zip             [('a', 'b')]    ['a'], ['b']
==============================================================
============================ main ============================
Position     Name            Type            Argtypes
--------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
1:9          zip             [('a', 'b')]    ['a'], ['b']
10:11        ints            ['Int']         None
0:0          print           'Void'          't'
12:18        zipped          [('Int', 'Bool')] None
8:5          main            'Int'
11:12        bools           ['Bool']        None
==============================================================
============================ zip =============================
Position     Name            Type            Argtypes
--------------------------------------------------------------
1:18         a               ['a']           None
1:24         b               ['b']           None
0:0          isEmpty         'Bool'          ['t']
1:9          zip             [('a', 'b')]    ['a'], ['b']
0:0          print           'Void'          't'
8:5          main            'Int'
==============================================================
```

7.9 Pretty printer test

The next example is more aimed at the parser and pretty printer than the type checker: we want to demonstrate that our parser is working, and that the pretty printer will print our code in a correct way without unneeded braces. It does also test the type checker, though, as the types of all operators need to

be correct for this to pass. This code can be found in `prettyprinttest.spl`.

```
1 Int calc(Int x, Int y, Int z) { if((y+1)>0 && x / (y+1) > z) {
2 return (((((5*(((2+x)-6)+z))+(42%3))))); } return 0; }
3
4 Int main() {
5     return calc(1,2,3);
6 }
```

When we omit one brace at the end, our parser will complain:

```
Exception: [Line 2:40] Expected ')' or ',', but got: ;
```

When we compile this program, we get the following symbol table and pretty-printed code (which is much better readable):

```
Int calc(Int x, Int y, Int z)
{
        if ((((y + 1) > 0) && ((x / (y + 1)) > z)))
        {
                return ((5 * (((2 + x) - 6) + z)) + (42 % 3));
        }
        return 0;
}
Int main()
{
        return calc(1, 2, 3);
}

========================== _glob ============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
4:5          main            'Int'
1:5          calc            'Int'           'Int', 'Int', 'Int'
0:0          isEmpty         'Bool'          ['t']
=============================================================
========================== calc =============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
1:21         y               'Int'           None
4:5          main            'Int'
0:0          isEmpty         'Bool'          ['t']
1:28         z               'Int'           None
0:0          print           'Void'          't'
1:14         x               'Int'           None
1:5          calc            'Int'           'Int', 'Int', 'Int'
=============================================================
========================== main =============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
1:5          calc            'Int'           'Int', 'Int', 'Int'
4:5          main            'Int'
0:0          isEmpty         'Bool'          ['t']
=============================================================
```

7.10 Shadowing

Functions and variables cannot have the same id, since this would create ambiguity. This is demonstrated with the program found in `shadowing.spl`.

```
1  Int x = 42;
2
3  Int sum([Int] xs) {
4      if (isEmpty(xs)) {
5          return 0;
6      }
7      return xs.hd + sum(xs.tl);
8  }
9
10 Void main() {
11     Bool x = True; // works, but produces a warning: global x unreachable
12     Int sum = 5;
13     // Int dum = sum(1 : 2 : 3 : []); //no longer works: sum is shadowed
14     return;
15 }
```

The statement on line 13 will thus give an exception, since *sum* is a var here:

```
Exception: [Line 13:12] 'sum' is a variable, not a function.
```

When we uncomment the statements on line 11 and 12, we will get an warning about a redefinition:

```
[Line 11:7] Warning: redefinition of global x
[Line 1:5] Previous definition was here
[Line 12:6] Warning: redefinition of global sum
[Line 3:5] Previous definition was here
```

But this is not an error: this program will still pass the type checking test and pretty-print fine:

```
Int x = 42;
Int sum([Int] xs)
{
        if (isEmpty(xs))
        {
                return 0;
        }
        return (xs.hd + sum(xs.tl));
}
Void main()
{
        Bool x = True;
        Int sum = 5;
        return;
}
```

```
============================ _glob ============================
Position     Name            Type            Argtypes
-------------------------------------------------------------
0:0          print           'Void'          't'
1:5          x               'Int'           None
3:5          sum             'Int'           ['Int']
10:6         main            'Void'
0:0          isEmpty         'Bool'          ['t']
```

```
============================================================
============================ sum ===========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
3:15         xs              ['Int']         None
0:0          print           'Void'          't'
1:5          x               'Int'           None
3:5          sum             'Int'           ['Int']
10:6         main            'Void'
============================================================
============================ main ==========================
Position     Name            Type            Argtypes
------------------------------------------------------------
0:0          isEmpty         'Bool'          ['t']
0:0          print           'Void'          't'
11:7         x               'Bool'          None
12:6         sum             'Int'           None
10:6         main            'Void'
============================================================
```

8 Division of work

8.1 Part 2: Semantical analysis

As with the previous part, we did most of the work together, at the university. As our schedules are quite in sync, it is relatively easy to get together and work on the compiler for hours at an end. We got most of the code done this way. The week prior to the deadline, we had an extremely busy week due to the Cryptography 2 exam (as the TU/e has their exams a week later). This resulted in having to squeeze in a few hours whenever possible, which ended up cumulating in staying late at the TU/e after the exam to finish up.

The only part we really did do parallel rather than together was the creation of this document, as it can clearly be 'multithreaded'. Formulating and discussing typing rules and example programs that showed the capabilities of our type checker was quite straight forward, and we both did our part while staying in touch via IRC.

This way of working worked out well for us, and generally keeps us on schedule, as it forces us to plan regular 'sessions' to work on the compiler. We aim to keep this up during the last two parts of the course.

8.2 Part 3: Code Generation

Similarly to the previous parts, we again worked on the code generation together rather than actually dividing the work. In our experience, part 3 was conceptually a bit more difficult to get started with. This caused us to first spend a few hours mapping out our plans, defining how we would translate function calls, and choosing how we would store the data structures. It was also valuable to just try out various (combinations of) assembly instructions in the interpreter and discuss the results. As we had quite a thorough plan (and understanding of the language) before we started writing the generator, writing the actual code generator was surprisingly easy.

As the deadline for this part was quite lenient, we were able to finish it all (i.e. including this document) while working together and discussing the code constantly as we progressed. This proved to be a great way to prevent errors and inconsistencies.

This way of 'dividing' the work seems to work well for us, so we intend to approach part 4 in a similar fashion.