Lecture Notes for DM565

Kim Skak Larsen

October 5, 2023

Introduction

This note contains material for the course DM565, Fall 2023. We discuss scopes, symbol tables, type checking, and code generation.

Scopes

Usually, portions of source code form a hierarchy, and we will use “scope” to refer to one component of such a hierarchy, which is usually thought of as a maximal portion of the code where the environment, in some sense, is the same.

The purpose of scopes is to encapsulate identifiers (sometimes called “names”) so that identifiers from one part of the source code does not influence logically separate parts of the program. This also means that the same identifiers can be used in different scopes, which is a convenience, and when using recursion, a necessity.

Usually some visual mechanism is defined to trigger a new scope. In C, a block surrounded by curly brackets defines a new scope. Also record-like structures (structs), class definitions, methods, etc. define new scopes. A programming language allows *nested scopes* if scopes can be introduced inside other scopes.

Consider the small SCIL program of Figure 1. In this programming language curly brackets do *not* define a new scope, but a function definition does. Thus, in that example, there are four scopes, one for each function and one for the main program.

In each scope, we are allowed to use the same identifiers as in other scopes. If we do that, which identifier we refer to is defined by the hierarchy. For example, when we refer to `a` in `h`, we refer to the “nearest”, which is the one declared in `g`. This means that when we get out to the scope of `f` again, its version of `a` (the 5th

```
var a, dummy

function f(){

  function g(){

    var a

    function h(){
      a = 0;
      print(a); # 3rd output: 0
    }

    a = 2;
    print(a); # 2nd output: 2
    dummy = h();
    print(a); # 4th output: 0
    return 0;
  }

  print(a); # 1st output: 1
  dummy = g();
  print(a); # 5th output: 1
  return 0;
}

a = 1;
dummy = f();
print(a); # 6th output: 1
return 0;
```

Figure 1: Static, nested scope example in SCIL.

output) is unchanged. Note that “nearest” is not a textual reference, but refers to the hierarchy of scopes.

All together, this scoping mechanism, respecting hierarchies, is referred to as *static, nested scoping*. “Nested” refers to the hierarchy and “static” to the fact that we can determine where an identifier “lives” from the source code. Some programming languages have a more dynamic (determined by the concrete execution of the program) definition of when and where identifiers are accessible.

The illustrations of Fig. 2 concern a hierarchy of named, nested functions, and show which names are visible at different locations. In Fig. 2a, note that the *name* of a function belongs to a scope further out than the body of the function. Function parameters (not shown) belong to the scope of the body. Fig. 2b shows old-fashioned (one-pass) scoping rules as they are known from C. In Fig. 2c, the indexed A’s are all variables named A; indices are just there for reference so we can indicate which A is visible. Finally, Fig. 2d demonstrates that as we enter and leave scopes, going from top to bottom in some source code, the collection of visible names behave like a stack.

Symbol Tables

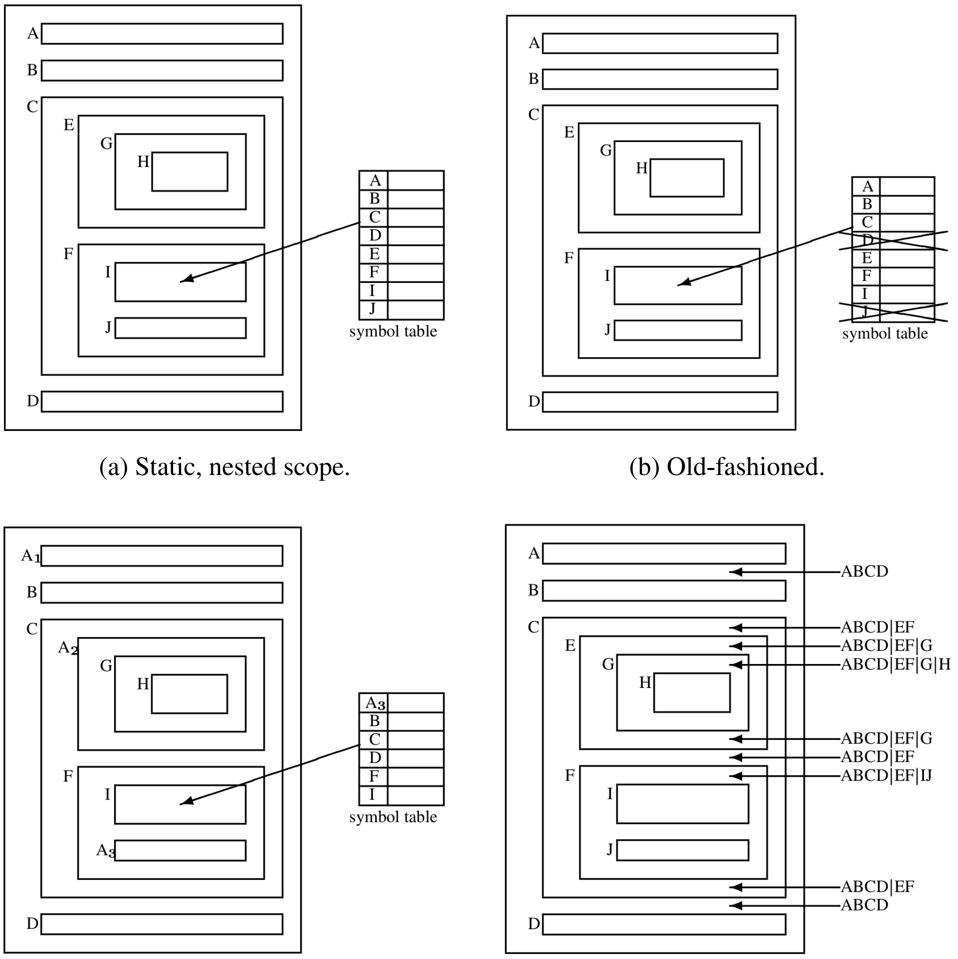
To keep track of where names are available, which instance of a name we are talking about, and where it may be found, we collect this information and organize it into what is called a *symbol table*.

One could just use a dictionary (a red-black tree or some other implementation), and as we progress through the code we can add names as we enter an inner scope and remove them again when leaving, as illustrated in Fig. 2d. One can also use some kind of versioning (persistency) to efficiently maintain all the different symbol tables in one structure.

We suggest another approach that closely follows the hierarchical structure of the source code. First, we discuss how this could be done in C, and then what one would naturally do in Python.

For C, the symbol table is constructed as a collection of hash tables, connected as an inverted tree; see Fig. 3. The entire construction illustrates the symbol table and each box illustrates a hash table. Fig. 4 shows a possible header file for this structure.

The elements in the symbol table are strings, `name`, with an associated value field, `value`. When such an element is inserted into the symbol table, it is stored into one of the hash tables. This will be described in detail below.



(a) Static, nested scope.

(b) Old-fashioned.

(c) Name hiding.

(d) Stack behavior.

Figure 2: Static, nested scope examples.

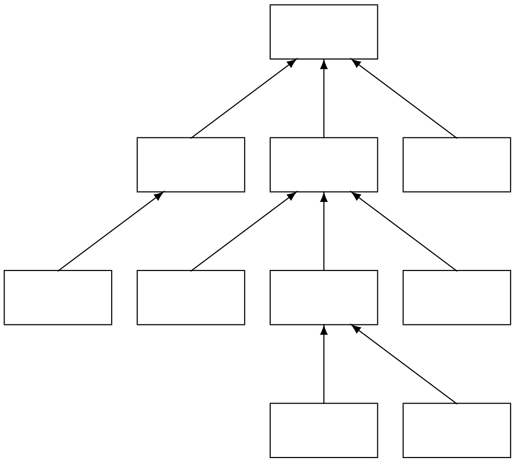


Figure 3: Overall illustration of a symbol table organization.

```
#define HashSize 317
#define NEW(type) (type *)malloc(sizeof(type))
void *malloc(unsigned n);

typedef struct SYMBOL {
  char *name;
  int value;
  struct SYMBOL *next;
} SYMBOL;

typedef struct SymbolTable {
    SYMBOL *table[HashSize];
    struct SymbolTable *next;
} SymbolTable;

int Hash(char *str);

SymbolTable *initSymbolTable(SymbolTable *t);

SYMBOL *insert(SymbolTable *t, char *name, int value);

SYMBOL *lookup(SymbolTable *t, char *name);
```

Figure 4: Header file for a symbol table implementation in C.

A pointer to a hash table can also be thought of as a pointer to (parts of) the symbol table which can be accessed through the pointer to the given hash table.

Usually, the semantics is that variables defined in a given scope can also be seen (accessed) in inner scopes; thus, the inverted tree structure is perfect for emulating this.

Each hash table will be used to store the names within one scope, and given a pointer to a hash table, one can access all the hash tables closer to the root, corresponding to all the scopes surrounding the one currently under discussion. Thus, the hash table in the root corresponds to the scope of the main program.

Conflicts during insertions into the hash tables are resolved using chaining. Thus, the entries in the hash table arrays are (possibly empty) linked lists of elements of the type `SYMBOL` (linked via `SYMBOL`'s `next` field). For each `name`, there is a value of type `SYMBOL` in which `name` is stored. To avoid any confusion, chaining is handled within each hash table and has nothing to do with the pointers seen in Fig. 3.

We now detail the functionality of the functions:

- `Hash` computes the hash values.
- `initSymbolTable` takes a pointer (possibly NULL) to a hash table `t` as argument and returns a new hash table with a pointer to `t` in its `next` field.
- `insert` takes a hash table and a string, `name`, as arguments and inserts `name` into the hash table together with the associated value `value`. A pointer to the `SYMBOL` value which stores `name` is returned.
- `lookup` takes a hash table and a string `name` as arguments, and it searches for `name` in the following manner: First search for `name` in the hash table which is one of the arguments of the function call. If `name` is not there, continue the search in the next hash table. This process is repeated recursively. If `name` has not been found after the root of the tree (see Fig. 3) has been checked, the result `NULL` is returned. If `name` is found, return a pointer to the `SYMBOL` value in which `name` is stored.

In Python, we would more naturally just use the built-in dictionary instead of our own hash table, and the entire implementation is given in Fig. 5.

```
class SymbolTable:
    """Implements a classic symbol table for static nested
       scope. Names for each scope are collected in a
       Python dictionary. The parent scope can be accessed
       via the parent reference.
    """
    def __init__(self, parent):
        self._tab = {}
        self.parent = parent

    def insert(self, name, value):
        self._tab[name] = value

    def lookup(self, name):
        if name in self._tab:
            return self._tab[name]
        elif self.parent:
            return self.parent.lookup(name)
        else:
            return None
```

Figure 5: A symbol table implementation in Python.

Symbol Collection

In a compiler, once the abstract syntax tree (AST) has been build, the next phase is usually symbol collection. This phase implements a recursive traversal of the AST, where all identifiers are located, and the name and associated information is inserted into the symbol table. The associated information is typically type information in a broad sense, i.e., we register if a variable name is an integer, a Boolean, a function, or something else. In languages with type definitions, type names and their definitions are also collected.

Subsequent phases that traverse the AST now also have the symbol table at their disposal. This is useful in, for instance, the type checking phase, since the location where a variable is used can be far from where it is defined. Thus, it is convenient to have a structure where one can look up relevant information about an identifier.

Type Checking

Type Checking takes place in a phase usually following the symbol collection phase. This phase is also a traversal of the AST, where one wants to check each operator, or more generally each language construction, to see if the arguments fulfill the requirements for the given operator to be applied.

One can define type correctness in many ways. One requirement could be that a variable declared to be of a certain type should never be assigned another type (at run-time). Using that philosophy, Program 1 should possibly be considered correct.

Program 1 Statically incorrect program that is dynamically correct because some code it not executed.

```
int i
i = 42
if False then
    i = "Hello World!"
```

In languages without explicit type declaration, one could decide that as long as operators are always applied to values of the correct type (still at run-time), then the program is type correct. In that case, Program 2 should be accepted. This kind of type correctness is based on the notion that a program is type correct if no type error will occur when we execute it. In general, it is not possible to decide this at compile-time, as one can see from Program 3, where the condition is provably undecidable.

Program 2 Statically incorrect program where dynamic correctness has been ensured through the control flow.

```
if “MyCondition” then
    v = 1
else
    v = "1"
...
if “MyCondition” then
    v = 10 * x
else
    v.append("0")
```

Partially for these reasons, we often decide to use a more restrictive form of type correctness, called *static type correctness*, where the requirements are phrased as

Program 3 Statically incorrect program that may or may not be dynamically correct.

```
i = 42
n = "read from input"
while "the n'th Turing machine halts on input n" do
    "something"
i = "Hello World!"
```

local rules that will guarantee the global guarantee that we should not encounter a type error at run-time. Though being restrictive, important advantages are that it is easy to understand and fairly easy to decide efficiently at compile-time (which is what the word *static* refers to).

We use the type declarations to determine the intended type of any variable, and then we have rules for all programming constructs. Thus, for "+", we would have a rule stating the type requirements for the operands, and what the resulting type of a "+"-expression is, so that this information can be used if the "+"-expression appears as a subexpression of a larger expression.

Similarly, we have local rules stating which conditions should be fulfilled when a function is called, when we index into an array, apply a length-operator to a variable, etc. Some examples are given in Fig 6, where we have sacrificed generality for readability.

A lot of concepts have been introduced to allow and explain systems with more flexibility that the strict rules above.

If we have both an integer and a float type, we may allow expressions such as `i + 42` and `x + 3.14`. We say that the operator "+" is *overloaded*, since it accepts more than one combination of types as operands. Overloading is not restricted to operations that are in some sense similar (such as addition); one could also overload by allowing "+" to mean string concatenation, as in `s + " my addition"`.

Sometimes we are allowed even more freedom. Our basic rules may say that we can add integers and we can add floats, but what if one writes `42 + 3.14`? We see this all the time in programming languages, but formally this is a *coercion*. What most languages accept here is that `42` is coerced into the floating point value `42.0`, then the operands are added, and the result is a float. There are no restrictions, other than what the programming language designer thinks is convenient, i.e., one could choose to allow `"Test " + 42`, defining that `42` is coerced to `"42"`, and then the operation computes the result `"Test 42"`.

In some languages, or just in some situations, the language designer will allow

⟨expression1⟩ + ⟨expression2⟩
Requirement: ⟨expression1⟩ and ⟨expression2⟩ must be of type integer
Resulting in: integer

`x` = ⟨expression⟩
Requirement: ⟨expression⟩ must be of the type that `x` is declared as

`f(`⟨expression1⟩`, ..., `⟨expression$k$⟩`)`
Requirement: `f` must be declared as a function with k parameters;
⟨expression$i$⟩ must be of the declared type of the ith
parameter of `f`

return ⟨expression⟩
Requirement: Let `f` be the immediately enclosing function;
⟨expression⟩ must of the declared return type of `f`.

Figure 6: Example type checking templates.

these things, but thinks it is best that the programmer declares to be fully aware of what is happening. When the programmer explicitly states that one type should be changed to another, this is referred to as a *cast*. So, one might have to write `"Test " + (string) 42`, and not including the cast would then typically result in a compile-time error.

Similar to the coercion discussion earlier, one can consider type mismatch at the time of assignment, as exemplified in Program 4. Many programming languages will decide in favor of allowing the first assignment. This special form of coercion is usually referred to as *assignment compatibility*. The next assignment is more controversial, but anything is possible and one could just decide that `False` and `True` are converted to the integers 0 and 1, for instance.

Program 4 Assignment compatibility issues.

```
int i
float x
x = 42
i = False
```

Finally, we discuss the concept of type *equivalence*. Consider Program 5, where we assume that one can define and name new types. One can apply a strict name-

Program 5 Type equivalence.

```
type apples =
    int
type oranges =
    int
```

based view on this, deciding that since we have defined two new types, they should never be mixed, i.e., never compared and never assigned to each other. Another view is to decide that types are equivalent if they contain exactly the same values.

The latter view on the problem is called *structural equivalence*. Many of these simple concepts become more interesting when discussing objects, or just records (structs in C), which can be considered very simple forms of objects, and the structural issues become clearer.

A first, very simple problem is illustrated in Program 6. It seems reasonable to define T1 and T2 to be structurally equivalent, but one has to think carefully about how to implement this. What do we do in our compiler implementation if we try to access the field b on some record, but we do not know if the record is of type T1 or T2? Think about how we could end up in a situation where, at compile-time, we have no idea if a record is of one type or the other.

Program 6 The definition of structural equivalence.

```
type T1 =
    record
        int a
        int b
    end
type T2 =
    record
        int b
        int a
    end
```

In Program 7, we have to decide if we allow any assignment compatibility between r1 and r2? There are languages that allow only r1 = r2 and languages that allow only r2 = r1, giving rise to different advantages and problems.

For type equivalence, many interesting questions turn up, as illustrated in Program 8. If one has the view that types should be equivalent when they contain the same values, then T1 and T2 should be equivalent.

Program 7 Assignment compatibility of records.

```
type T1 =
    record
        int a
        int b
        int c
    end
type T2 =
    record
        int a
        int c
    end
T1 r1
T2 r2
```

The problems become even more interesting when types may be used recursively. Consider Program 9, where both T1 and T2 can be used for making linked lists, and they are equivalent. In fact, they would also be equivalent if the occurrence of T2 inside the definition of T2 was replaced by T1.

One can also have mutually recursive types and then it gets increasingly difficult for a compiler to decide if types are equivalent. However, viewing these types as a DFA, one can design tailor-made algorithms for comparing two types, or use tools from automata theory to check for equivalence of two different starting states in such an automata. DFA minimization [3, 2, 1] is a useful tool since a minimal DFA is unique up to renaming of states, so it is easy to check equivalence of two DFAs once they are minimized. Hopcroft's algorithm for this has time complexity $O(|\Sigma| n \log n)$, where n is the number of states and Σ is the alphabet (here that would be the variable names in the records), so this can be done very fast.

Drawing the type information from Fig. 9, one arrives at the DFAs of Fig. 7, showing that if one has a record, r, of type T1, for example, then r.n.n.n.k is potentially allowed (assuming that the value exists). Running DFA minimization on these two automata, they become identical (the one for T1 is minimal).

As a special case of recursive type definitions, one has to decide if the types in Program 10 should be allowed or not. In either case, one has to detect it.

Program 8 Assignment compatibility of records.

```
type T1 =
    record
        int a
        record
            int b
            bool c
        end n
    end
type Help =
    record
        int b
        bool c
    end
type T2 =
    record
        int a
        Help n
    end
T1 r1
T2 r2
```

Code Generation

For a full, professional compiler, one would recommend to define a so-called *intermediate representation (IR)*. Compilation would then be done in two steps from the high level language to the IR, and then from the IR to executable code.

The advantages of that approach are that

- one can hook other high level languages up to the same IR (many languages have hooked-up to java byte code),
- one does not have to write complete compilers for each possible platform the high level language should work on; only separate translations from the IR to each platform, and
- it gives possibilities for carrying out various types of optimization before the code is emitted in its final form.

Due to time constraints we will not use an IR. Instead, we discuss a quite direct

Program 9 Recursive types.

```
type T1 =
    record
        int k
        T1 n
    end
type T2 =
    record
        int k
        record
            int k
            T2 n
        end n
    end
T1 r1
T2 r2
```

Program 10 Recursive types without basis.

```
type T1 =
    T2
type T2 =
    T3
type T1 =
    T1
```

translation from the abstract syntax tree (AST) to finished assembler.

A helpful approach in order to write structured and correct code is that of formulating *invariants* for the translation of constructs.

- Without formulating this very formally, invariants are conditions one can rely one throughout the compilation (code generation) process.
- We establish/maintain invariants by following protocols in various situations.
- Work has to be done to ensure that all constructs adhere to the protocols.
- The benefit is that it is much easier to implement constructions when we know some invariants we can rely on.

Here are some of the important invariants used in the code generation for SCIL.

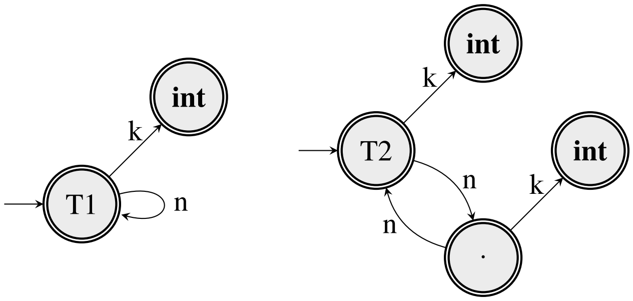
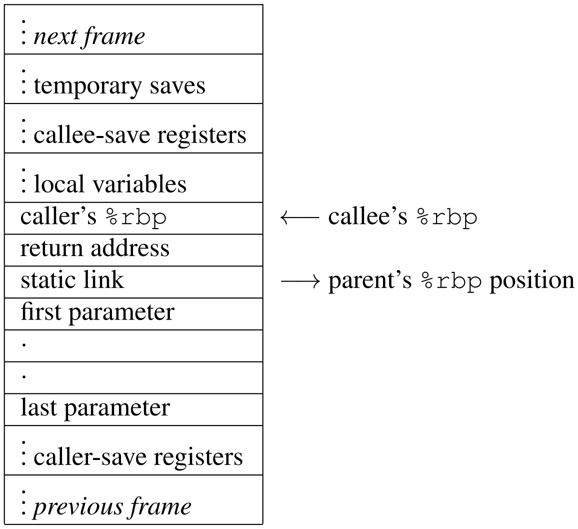


Figure 7: Example types as DFAs.

- The stack organization in the form of stack frames.
- Constructions "consuming" a number of operands pop them from the stack; the stack top is the last argument.
- Constructions "producing" a result push this onto the stack.
- For constructions with multiple operands, the operands are processed left-to-right.
- Function calls use the stack for all parameters.
- A function's return value is stored in `%rax`.
- We use 64 bits (8 bytes) for all types of values.

Some of these invariants were chosen for simplicity at some (fairly limited) cost in efficiency. For instance, using the stack for expression computation simplifies the code significantly, but comes at a small cost compared to utilizing registers more effectively.

We decide on the following stack frame (the format for the activation records).



Vertical dots indicated possible multiple entries. "Parent" refers to the stack frame for the latest function which is the immediate parent in the static nested scope.

We have included caller/callee-save registers, but in SCIL, we do not have active values in registers across function calls so we do not save any registers. They are included for generality, but listed in parenthesis below.

We will discuss the stack frame details further below, but first we list the protocols that must be followed when calling a function and when being called by another function.

To call another function, we carry out the following protocol. The part up until the function call is referred to as the Caller Prologue and the rest as the Caller Epilogue.

1. (push caller-save registers)
2. push parameters in reverse order
3. compute and push static link
4. call function [return value in `%rax`]
5. pop/deallocate static link and parameters
6. (pop/deallocate caller-save registers)

When being called, we carry out the following protocol. The part up until performing one's own computation is referred to as the Callee Prologue and the part after the computation as the Callee Epilogue.

1. `pushq %rbp`
2. `movq %rsp, %rbp`
3. allocate stack space for local variables
4. (push callee-save registers)
5. perform computation:
   - parameter 1 in `24(%rbp)`
   - return value in `%rax`
6. (pop callee-save registers)
7. deallocate local variables
8. `popq %rbp`
9. `ret`

Notice the stack-like behavior (parentheses-like structure).

On the basis of these invariants, we now consider code generation for a number of representative example constructions.

Addition expressions

⟨expression1⟩ + ⟨expression2⟩

The following is the code generation template for this.

```
code for ⟨expression1⟩
code for ⟨expression2⟩
popq reg1
popq reg2
addq reg1, reg2
pushq reg2
```

Note that, when we write something such as ⟨expression1⟩ + ⟨expression2⟩, this should just be read as a linearized form of the AST, i.e., this means we are considering a +-node in the AST. It has a left child, which is a (sub)AST representing the left-hand side expression, and it has a right child, which is a (sub)AST representing the right-hand side expression.

The translation process is organized as a collection of (mutually) recursive functions. Thus, when we write "code for ⟨expression1⟩" in the above, this means

"recursively generate code for the tree ⟨expression1⟩".

According to the invariants, we know where the arguments are, and we know we have to place the result on the stack top.

We use registers 1 and 2, but when emitting to assembler, they will of course be some of the available concrete registers.

In the following, we will need labels for the control-flow, such as "`else_part`". However, there are likely many `if`-statements in our code, and they should all have separate labels. This is handled by using a counter and appending unique values to the labels. We ignore this issue from now on.

If statements

if ⟨expression⟩ **then** ⟨statement1⟩ **else** ⟨statement2⟩

The template is the following.

```
            code for ⟨expression⟩
            popq reg1
            movq "true", reg2
            cmpq reg1, reg2
            jne else_part
            code for ⟨statement1⟩
            jmp end_if
else_part:
            code for ⟨statement2⟩
end_if:
```

As part of designing the compiler, one has to decide on the representation of values of all types. In particular, one has to decide how the Boolean value, "true", is represented. That is also a (simple) invariant. For readability, we simply write "true", but one should remember that this is a concrete value, which should be replaced by something else in the actual compiler (the integer "1", for instance).

While statements

while ⟨expression⟩ **do** ⟨statement⟩

```
while_start:
               code for ⟨expression⟩
               popq reg1
               movq “true”, reg2
               cmpq reg1, reg2
               jne while_end
               code for ⟨statement⟩
               jmp while_start
while_end:
```

Function definitions

Code must be generated according to the stack frame convention. Function labels are all produced in advance and remembered, since local functions use the same global counter for label uniqueness. Examples can be found in the published examples such as `factorial.s`. Essentially, the function definitions should just use the protocols.

```
               code for local functions
func_start:
               code for callee prologue
               allocate stack space for local variables
               code for function body
func_end:
               code for callee epilogue
```

Return statements

return ⟨expression⟩

We assume that func_end is the end-label of the nearest enclosing function for the **return**-statement. To have access to this requires that we establish a new invariant, namely that we maintain a stack of all the statically nested functions that we are currently in.

```
code for ⟨expression⟩
popq %rax
jmp func_end
```

Which register to use for the return value (here we have used `%rax` as an example) depends on the requirements or conventions of the target language, and we have listed this as one of our invariants.

Integer expression

integer

Just dealing with a constant integer is easy. It is an expression with no operands, so it consumes nothing, but the result should be pushed to stack, i.e., the template is

```
pushq integer
```

Here, **integer** is of course the concrete value, possible with extra syntax as required by the assembler language, e.g., `pushq $42`.

Identifier expression

ident

While an integer constant was the easiest template, the template for **ident** is one of the harder ones. Again, we just need to push the value, but it is somewhat complicated to determine the location of the value.

```
movq regBP, regSL
repeat level_difference times:
   movq -2(regSL), regSL
pushq offset(regSL)
```

The level_difference is computed as the different between the current level (of static, nested scope) and the level of **ident** (which is found by looking **ident** up in the symbol table).

We use mnemonic names for the registers used for this. The “-2” comes from the stack frame where static link is saved 2 words away from the base pointer. Using 64 bit assembler with bytes as the smallest addressable unit, the "-2" will be emitted as "-16" (2 times 8 bytes).

The offset is another piece of information that will be returned when searching for **ident** in the symbol table. The offset must be negated if the identifier is a formal parameter to a function (as opposed to a local variable) and further adjusted depending on how many words there are between the base pointer and where the parameters or local variables start.

Similar complications are involved when setting up the static link in the caller prologue.

We finish the section with an example discussion of something that is not part of SCIL, namely arrays.

Space allocation statements

allocate ⟨id-expression⟩ **of length** ⟨expression⟩

This is a generic form of memory allocation from the heap, similar to **malloc** in C. In some programming languages, one has the freedom to choose *not* to check for different error conditions; in other languages, checks are an obligatory part of the language specification.

So what is a heap? It is a part of memory, different from the stack, where one can store data. In C, **malloc** is simply the keyword saying use the heap instead of the stack. Now that we know how a stack works, we can see the reason. If we write a function for adding a leaf to a binary tree, we do not want the leaf to be deallocated when the function returns. Thus, structs, arrays, objects, etc. are often placed on the heap. In many languages, this is implicit, i.e., there is no special keyword. It is just a rule that if we instantiate an object, it is placed on the heap.

A heap can be organized in many ways; some heap organizations include a heap counter that indicated the next available piece of memory.

A somewhat abstract template for this is the following, communicating the idea that the variable (of type struct, object, or similar) is set to point to the next available heap space, and the heap counter is updated to reflect the used space.

```
code for ⟨expression⟩
(code for out-of-memory check)
movq "heap-counter", "address of ⟨id-expression⟩"
addq "⟨expression⟩-result", "heap-counter"
```

Index expressions

⟨id-expression⟩ **[** ⟨expression⟩ **]**

Not all indexing into an array is as simply as `A[42]`, for instance. One could have constructions as

```
B.values[i].sequence[f(x) + delta],
```

for example, where

⟨id-expression⟩	is	`B.values[i].sequence`, and
⟨expression⟩	is	`f(x) + delta`.

A somewhat abstract template could be the following.

code for ⟨expression⟩
"look up/compute address of ⟨id-expression⟩"
(code for range checks)
"compute final address"

In concluding this section, we emphasize that the overall important aspect is that the templates implement the correct semantic behavior of the programming languages we are implementing a compiler for. Thus, there are often many different correct templates one could choose from. Some, but not many, efficiency issues come into play when choosing. The reason why it is not that many is that giving the templates a logical, maintainable structure is important, and many aspects can be optimized in a later phase, independent of how efficiently the template was coded originally.

References

[1] Norbert Blum. An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57(2):65–59, 1996.

[2] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *International Symposium on the Theory of Machines and Computations*, pages 189–196, 1971.

[3] Wikipedia. DFA minimization, 2008. [Accessed October 4, 2019].