Supplementary Notes for DM546

Kim Skak Larsen

February 11, 2015

Introduction

This note contains supplementary material for the course DM546, Spring 2015, in relation to the course textbook [1]. The topics below are all covered to some extent in the textbook, but these notes present alternative or supplementary definitions and methods. Thus, these notes are brief and do not replace the material in the textbook.

Symbol Tables

The textbook suggests two possible implementations. Here we suggest a third. The symbol table is constructed as a collection of hash tables, connected as an inverted tree; see Fig. 1. The entire construction illustrates the symbol table and each box illustrates a hash table. Fig. 2 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 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.

The intended use is for a programming language with nested scopes. Here, a scope can be thought of as an area of the textual program where variables and other names of entities are defined. The mechanisms for defining a scope vary greatly from language to language, but usually function or method definitions introduce a new scope, but sometimes defining a block (in many languages represented by curly brackets) is sufficient. Similarly, record-like structures such as structs from
C may introduce a new scope, or it may just be convenient to treat it as such. A programming language allows nested scopes if scopes can be introduced inside other scopes. An example from C can be seen in Figure 3. 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. 1.

We now detail the functionality of the six functions.

- **Hash** computes the hash values.
- **initSymbolTable** returns a pointer to a new initialized hash table (of type `SymbolTable`).
- **scopeSymbolTable** takes a pointer to a hash table `t` as argument and
Figure 2: The file `symbol.h`. 

```c
#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 *scopeSymbolTable(SymbolTable *t);
SYMBOL *putSymbol(SymbolTable *t, char *name, int value);
SYMBOL *getSymbol(SymbolTable *t, char *name);
```
#include <stdio.h>

int i = 1;  /* i defined at file scope */

int main(int argc, char * argv[])
{
    printf("%d\n", i);  /* prints 1 */

    {
        int i = 2, j = 3;  /* i, j defined at block scope */
        printf("%d\n%d\n", i, j);  /* prints 2 and 3 */

        {
            int i = 0;  /* i defined in nested block */
            printf("%d\n%d\n", i, j);  /* prints 0 and 3 */
        }
    }

    printf("%d\n", i);  /* prints 2 */

}

printf("%d\n", i);  /* prints 1 */

return 0;

Figure 3: An example of nested scopes in C.
returns a new hash table with a pointer to t in its next field.

- putSymbol 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.

- getSymbol 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. 1) has been checked, the result NULL is returned. If name is found, return a pointer to the SYMBOL value in which name is stored.

Type Checking

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.

```
1: int i
2: i = 42
3: if False then
4:   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.

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>if “MyCondition” then</td>
</tr>
<tr>
<td>2:</td>
<td>v = 1</td>
</tr>
<tr>
<td>3:</td>
<td>else</td>
</tr>
<tr>
<td>4:</td>
<td>v = &quot;1&quot;</td>
</tr>
<tr>
<td>5:</td>
<td>...</td>
</tr>
<tr>
<td>6:</td>
<td>if “MyCondition” then</td>
</tr>
<tr>
<td>7:</td>
<td>v = 10 * x</td>
</tr>
<tr>
<td>8:</td>
<td>else</td>
</tr>
<tr>
<td>9:</td>
<td>v.append(&quot;0&quot;)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>i = 42</td>
</tr>
<tr>
<td>2:</td>
<td>n = “read from input”</td>
</tr>
<tr>
<td>3:</td>
<td>while “the n’th Turing machine halts on input n” do</td>
</tr>
<tr>
<td>4:</td>
<td>“something”</td>
</tr>
<tr>
<td>5:</td>
<td>i = &quot;Hello World!&quot;</td>
</tr>
</tbody>
</table>

As explained in the textbook, 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 4, where we have sacrificed generality for readability.

Some concepts that are not all made explicit in the textbook at the best possible location are the following:

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
\langle expression1 \rangle + \langle expression2 \rangle

Requirement: \langle expression1 \rangle and \langle expression2 \rangle must be of type integer

Resulting in: integer

\times = \langle expression \rangle

Requirement: \langle expression \rangle must be of the type that \times is declared as

f (\langle expression1 \rangle, \ldots, \langle expressionk \rangle)

Requirement: f must be declared as a function with k parameters;
\langle expressioni \rangle must be of the declared type of the ith parameter of f

return \langle expression \rangle

Requirement: Let f be the immediately enclosing function;
\langle expression \rangle must of the declared return type of f.

Figure 4: Example code generation templates.

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 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.

Finally, we discuss the concept of type equivalence. Consider Program 5, where
Program 4 Assignment compatibility issues.

1: int i
2: float x
3: x = 42
4: i = False

we assume that one can define and name new types. One can apply a strict name-

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.

This 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.

In Program 6, 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

Program 5 Type equivalence.

1: type apples =
2:     int
3: type oranges =
4:     int

Program 6 Assignment compatibility of records.

1: type T1 =
2:     record
3:         int a
4:         int b
5:         int c
6:     end
7: type T2 =
8:     record
9:         int a
10:        int c
11:    end
12: T1 r1
13: T2 r2
allow only $r_2 = r_1$, giving rise to different advantages and problems. For type equivalence, many interesting questions turn up, as illustrated in Program 7. If one has the view that types should be equivalent when they contain the

Program 7 Assignment compatibility of records.

```plaintext
1: type T1 =
2:    record
3:      int a
4:    record
5:      int b
6:    bool c
7: end n
8: end
9: type Help =
10: record
11:    int b
12:    bool c
13: end
14: type T2 =
15:    record
16:      int a
17: Help n
18: end
19: T1 r1
20: T2 r2
```

same values, then $T_1$ and $T_2$ should be equivalent.

The problems become even more interesting when types may be used recursively. Consider Program 8, where both $T_1$ and $T_2$ can be used for making linked lists, and they are equivalent. In fact, they would also be equivalent if the occurrence of $T_2$ inside the definition of $T_2$ was replaced by $T_1$.

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 [4, 3, 2] 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 $\Sigma$ is the alphabet (here that
Program 8 Recursive types.
1: type T1 =
2:   record
3:     int k
4:     T1 n
5:   end
6: type T2 =
7:   record
8:     int k
9:     record
10:    int k
11:    T2 n
12:   end n
13:   end
14: T1 r1
15: T2 r2

would be the variable names in the records), so this can be done very fast.
As a special case, one has to decide if the types in Program 9 should be allowed or not. In either case, one has to detect it.

Program 9 Recursive types without basis.
1: type T1 =
2:   T2
3: type T2 =
4:   T3
5: type T1 =
6:   T1

Code Generation

Instead of the more complicated approach in the textbook, we discuss a quite direct translation from the abstract syntax tree (AST) to (almost) finished assembler. The only unfinished part has to do with placing variables in registers, on the stack, etc. We generate the code using what is called temporary variables, and in a later pass, these are replaced by registers or actual memory addresses. The reason for this is that the utilization of registers is very important and we would like the opportunity
to have a separate phase devoted to that task. In the code generation phase, we
do not reuse temporary variables, but instead assume that we can use as many as
we want. The code we generate is actually so close to being finished that if we
use temporaries \( t_1, \ldots, t_n \), we could allocate \( n \) words of memory and use the \( i \)th
entry every time we refer to \( t_i \), and the code should work (on a CPU allowing two
memory address instructions in one operation).

On that basis, we now consider code generation for the following:

**Addition expressions**

\[ \langle \text{expression1} \rangle + \langle \text{expression2} \rangle \]

code for \( \langle \text{expression1} \rangle \)

“place result in a temporary”

code for \( \langle \text{expression2} \rangle \)

“place result in another temporary”

“add temporaries and place the result in yet another temporary”

The translation process is organized as a collection of (mutually) recursive func-
tions. Thus, when we write “code for \( \langle \text{expression1} \rangle \)” in the above, this means

“recursively generate code for the tree \( \langle \text{expression1} \rangle \)”.

Note that one must decide on a convention for where one finds the result of an
expression computed by code generated recursively such that it is possible to move
the result into a (known) temporary.

Below, we will generate code for subexpressions in many places and the result will
have to reside in some temporary. This must be done according to the convention
one decides, and we ignore the details in the rest of this section.

We also 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.

We discuss a representative collection of code generation template examples in the
following:
If statements

```plaintext
if ⟨expression⟩ then ⟨statement1⟩ else ⟨statement2⟩
```

We are considering a direct translation from AST to executable assembler, so the line above should be read as a linearized form of a part of the AST, i.e., this is a if_then_else node with three children, representing ⟨expression⟩, ⟨statement1⟩, and ⟨statement2⟩. The template is the following:

```plaintext
code for ⟨expression⟩
cmp ”⟨expression⟩-result”, “true”
jne else_part
code for ⟨statement1⟩
jump 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. For readability, we simply write “true”, but one should remember that this is a concrete value (the integer “1”, for instance). The quotes are there to remind you that this part should be replaced by something else in an actual compiler.

Also, instead of just using temporaries \(t_1, t_2, \ldots\), we improve readability by using names, such as “⟨expression⟩-result”. However, this is just the next available \(t_i\) at this point in the compilation process.

While statements

```plaintext
while ⟨expression⟩ do ⟨statement⟩
```

```plaintext
while_start:
code for ⟨expression⟩
cmp ”⟨expression⟩-result”, “true”
jne while_end
code for ⟨statement⟩
jump while_start
while_end:
```
Space allocation statements

allocate \langle id-expression \rangle of length \langle expression \rangle

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.

code for \langle expression \rangle
(code for out-of-memory check)
  mov “heap-counter”, “address of \langle id-expression \rangle”
  add “\langle expression \rangle-result”, “heap-counter”

Index expressions

\langle id-expression \rangle [ \langle expression \rangle ]

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

\text{B.values[i].sequence[f(x) + delta]},

for example, where

\langle id-expression \rangle is \text{B.values[i].sequence}, and
\langle expression \rangle is f(x) + delta.

code for \langle expression \rangle
“look up/compute address of \langle id-expression \rangle”
(code for range checks)
“compute final address”

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. Details can be found in the published examples (factorial.s, for instance).
Return statements

return ⟨expression⟩

We assume that func_end is the end-label of the nearest enclosing function for the return statement.

code for ⟨expression⟩
mov "⟨expression⟩-result", %eax
jump func_end

Which register to use for the return value (here we have used %eax as an example) depends on the requirements or conventions of the target language.

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.

Peephole Optimization

A peephole optimizer can be characterized as follows: It

- works (most often) at the assembler code level;
- looks only at peepholes, which are sliding windows on the code sequence (rarely more than one window at a time);
- uses patterns to identify and replace inefficient constructions;
- continues until a global fixed point is reached; and
- usually optimizes for both time and space.
Some often used ideas are

- change $x = x + 1$ to `inc x`;
- use algebraic laws;
- utilize jump short-circuiting;
- control push/pop effects;
- focus on template interaction.

By $x = x + 1$, we mean the concrete variant in whichever target language is used and there could be many concrete formulations, depending on how the compiler is written and the style of the code generated. A possible pattern could be:

\[
\text{mov } t1, t2 \\
\text{add } $1, t2 \rightarrow \text{inc } t2, \text{ provided that } t2 \text{ is not used again} \\
\text{mov } t2, t1
\]

Algebraic laws cover patterns of the flavor:

\[
\begin{align*}
x \ast 0 & \rightarrow 0 \\
x \ast 1 & \rightarrow x \\
x \ast 2 & \rightarrow x + x
\end{align*}
\]

One always has to test and/or check specifications to know what is best. In the last example, for instance, addition is a faster operation than multiplication, so the transformation is probably good if $x$ is in a register. If $x$ is a memory address, will the CPU then retrieve it twice from a slow memory? Then it is probably worse. And if one is changing instructions anyway, then maybe `leftshift x` is even better.

A natural question is why we generate non-optimal code to begin with. The answer is that we do that for the sake of modularity. We probably did not write explicit code for generating $x \ast 1$. Instead, this was produced by general code that works for any constant. Instead of messing up that code, it is often better to use the peephole technique.

In designing templates for code generation, we of course try to make sensible ones, but our desire to keep things modular means that code at the end of one template and the beginning of another may not have been considered together in earlier
phases of the compiler. This can rise to push/pop patterns, meaning that a value is pushed at the end of one template just to be popped back to where it came from in the beginning of the next. Such an occurrence can just be deleted. Similarly, one may end a template with a jump, just to arrive at code from another template that starts with a jump. Obviously, we could fix this, replacing two consecutive template jumps by one.

Having defined a collection of patterns, we apply the algorithm in Fig. 5.

\[\text{repeat}\]
\[\quad\text{for each instruction in succession do}\]
\[\quad\quad\text{for each peephole pattern in succession do}\]
\[\quad\quad\quad\text{repeat}\]
\[\quad\quad\quad\quad\text{apply the peephole pattern to the code}\]
\[\quad\quad\quad\quad\text{until the code (goto graph) did not change}\]
\[\quad\quad\text{end}\]
\[\quad\text{end}\]
\[\text{until the code (goto graph) did not change}\]

**Figure 5:** The peephole algorithm

One should go through a validation process after, or better concurrently with, implementing peephole optimization and designing the patterns. One should

- argue that each peephole pattern preserves semantics;
- demonstrate that each peephole pattern is realized correctly;
- show using program statistics that the optimizer improves the programs;
- prove that the peephole optimizer terminates.

With regards to semantics, note that this includes that one is not allowed to “fix” the program, i.e., if the original program would terminate with an error, it should still do so afterwards, and if something was computed incorrectly due to overflow, it should have the same problem afterwards.

The reason that it is necessary to prove termination is that the algorithm contains a priori unbounded loops. One could imagine that combinations of replacing patterns by other ones would lead to an infinite loop.

Termination is proved using a termination function. This is a proof-technical construction (that is, one does not implement the termination function). A function is a
termination function if its domain is the set of all possible programs, its codomain is the natural numbers, and the function value decreases every time we apply a peephole pattern.

Clearly, if these conditions are fulfilled, this constitutes a proof of termination. The argument goes as follows: Evaluate the function on the original code to get some value \( s \). Since the function is integer-valued, when we change the code, the next value is \( s - 1 \) or smaller. Since the smallest value in the natural numbers is zero, at most \( s \) patterns can be applied.

A method that often works is to define the termination function as a sum of counts of different parts of the program, weighted appropriately. In this way, we automatically get that all function values are non-negative. Often used ingredients are code size, number of complex operations, length of jump chains, etc.

Here is an example: Assume the only pattern we have is one of the ones from above, where we replace three instructions by one. We define the function \( t(c) = \#\text{instructions} \), i.e., the number of assembler instructions in the code. Let \( c \) be the code before we make a change, and let \( c' \) be the code after one application of this pattern, then clearly \( t(c') = t(c) - 2 \), i.e., we have a decrease of at least one. This is a proof that with only that one pattern, our peephole algorithm terminates.

Now, we add the pattern

\[
\text{mul} \ 2, \ t \quad \rightarrow \quad \text{add} \ t, t
\]

When this pattern is applied, our current termination function does not decrease. Thus, we no longer have a proof of termination. We adjust the function to be \( t(c) = \#\text{instructions} + \#\text{mul} \), and we have a proof again, since when the new pattern is applied, a \text{mul}-operation is removed.

Adding the pattern,

\[
\text{add} \ 1, t \quad \rightarrow \quad \text{inc} \ t
\]

we could try the same approach, and define \( t(c) = \#\text{instructions} + \#\text{mul} + \#\text{add} \), but this does not work, since the second pattern replaces a \text{mul}- with an \text{add}- instruction.

Here, we can use weights, and instead define

\[
t(c) = \#\text{instructions} + 2\#\text{mul} + \#\text{add},
\]

and the application of any of the three patterns give rise to a decrease by at least one.

Note that one could imagine patterns that make the code longer, such as
add $2$, $t$ $\rightarrow$ inc $t$

which can also be handled by appropriate weighting.

**Other Important Optimization Ideas**

In earlier phases of the compiler, one can consider common subexpression detection and elimination, constant propagation, constant folding, loop unravelling, and function inlining, to mention some.

*Loop unravelling* can be illustrated by Program 10. Two ways of computing the result are shown and it is not clear what is best. Of course, one can unravel even more; sometimes completely removing the *for*-loop.

Another important technique is *function inlining*. Consider Program 11, where we have taken the body of the function and placed it directly instead of the function call, making the appropriate renaming of variables. This technique can potentially save a lot of expensive function calls, but it can be tricky to avoid clashes between variables from the calling and the called function, and one has to handle changes in the treatment of static links, etc.

**Acknowledgment**

Thanks to Nikolaj Nøjgaard for comments to an initial draft.
Program 11 Function inlining.

1: function double(x)
2:     return 2*x
3: i = 0
4: while i < 10 do
5:     y = double(y)
6:     i += 1
7: ...
8: i = 0
9: while i < 10 do
10:    y = 2*y
11:    i += 1

References


