# Supplementary Notes for DM546

Kim Skak Larsen

February 8, 2016

## Introduction

This note contains supplementary material for the course DM546, Spring 2016, 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

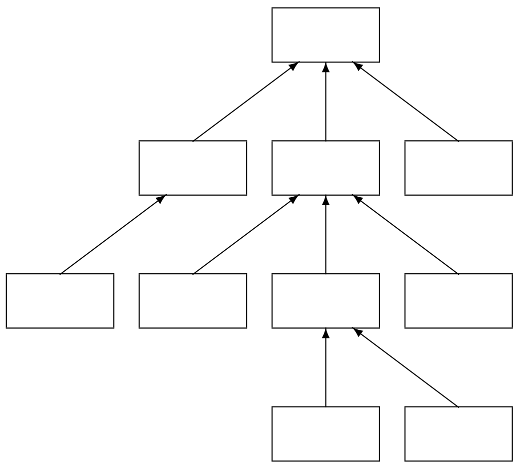

Figure 1: An example of connections between hash tables.

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

```
#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);
```

Figure 2: The file `symbol.h`.

```
#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.

```
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.

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.

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

**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!"
```

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

`x` = $\langle$expression$\rangle$
    *Requirement:* $\langle$expression$\rangle$ must be of the type that `x` is declared as

`f(`$\langle$expression1$\rangle$`, ..., `$\langle$expression$k\rangle$`)`
    *Requirement:* `f` must be declared as a function with $k$ parameters; $\langle$expression$i\rangle$ must be of the declared type of the $i$th 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.

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

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.

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.

A first, very simple problem is illustrated in Program 6. It seems reasonable to

**Program 6** The definition of structural equivalence.

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

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.

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

**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
```

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.

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 [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 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 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

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:

**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
```

**Addition expressions**

⟨expression1⟩ + ⟨expression2⟩

> code for ⟨expression1⟩
> "place result in a temporary"
> code for ⟨expression2⟩
> "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 functions. Thus, when we write "code for ⟨expression1⟩" in the above, this means

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

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

**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:

```
            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, which should be replaced by something else in an actual compiler (the integer "1", for instance).

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**

**while** ⟨expression⟩ **do** ⟨statement⟩

```
while_start:
            code for ⟨expression⟩
            cmp "⟨expression⟩-result", "true"
            jne while_end
            code for ⟨statement⟩
            jump while_start
while_end:
```

**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.

```
code for ⟨expression⟩
(code for out-of-memory check)
mov "heap-counter", "address of ⟨id-expression⟩"
add "⟨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`. |

```
code for ⟨expression⟩
“look up/compute address of ⟨id-expression⟩”
(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).

```
            code for local functions
func_start:
            code for variable declarations
            code for start-of-function
            code for function body
func_end:
            code for end-of-function
```

**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:

```
mov t1,t2
add $1,t2   ⟶   inc t2,   provided that t2 is not used again
mov t2,t1
```

Algebraic laws cover patterns of the flavor:

```
x * 0  ⟶  0
x * 1  ⟶  x
x * 2  ⟶  x + x
```

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 * 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 jumps by one.

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

```
repeat
    for each instruction in succession do
        for each peephole pattern in succession do
            repeat
                apply the peephole pattern to the code
            until the code (goto graph) did not change
        end
    end
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 $t(c) = \#\mathrm{instructions}$, 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

$$\textbf{mul}\ \texttt{2,t} \longrightarrow \textbf{add}\ \texttt{t,t}$$

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 **mul**-operation is removed.

Adding the pattern,

$$\textbf{add}\ \texttt{1,t} \longrightarrow \textbf{inc}\ \texttt{t}$$

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 **mul**- with an **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

$$\textbf{add}\ \texttt{2,t} \longrightarrow \begin{array}{l}\textbf{inc}\ \texttt{t}\\ \textbf{inc}\ \texttt{t}\end{array}$$

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 11. Two ways of computing the

**Program 11** Loop unravelling.

```
for i = 0; i < 2*N, i++ do
    A[i] = B[i] + C[i]
...
for i = 0; i < 2*N, i += 2 do
    j = i + 1
    A[i] = B[i] + C[i]
    A[j] = B[j] + C[j]
```

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 12, 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

**Program 12** Function inlining.

```
function double(x)
    return 2*x
i = 0
while i < 10 do
    y = double(y)
    i += 1
...
i = 0
while i < 10 do
    y = 2*y
    i += 1
```

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 on an initial draft.

## References

[1] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998. Reprinted with corrections, 1999; reissued, 2004.

[2] 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.

[3] 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.

[4] Wikipedia. DFA minimization, 2014. [Accessed January 26, 2015].