3.2 Names

```
structure Stack =
struct
 type elt = int
 datatype stack = ST of elt list ref
 val initStack = ST(ref[ ])
end

structure StackUser1 =
struct
 structure Stack1 = Stack
 ...
end

structure StackUser2 =
struct
 structure Stack2 = Stack
 ...
 datatype stack = ST of elt list ref
end
```

All the following sharing equations hold: *StackUser1*.*Stack1* = *StackUser2*.*Stack2*, *elt* = *int*, *Stack*.*stack* = *StackUser1*.*Stack1*.*stack*. None of the following sharing equations hold: *StackUser1* = *StackUser2*, *Stack*.*stack* = *StackUser2*.*stack*

Sharing equations can be decided by decorating programs with NAMES. There are two kinds:

structure names:	$n1, n2, \ldots,$
	$m1, m2, \ldots$
type names:	$t1, t2, \ldots,$
	$s1, s2, \ldots,$
	$unit, int, bool, \rightarrow$

Two structures SHARE if they are decorated by the same structure name; two types SHARE if they are decorated by the same type name.

3.3 Decorating Structures

Each elaboration of a structure expression of the form

```
struct ... end
```

yields a fresh structure, i.e. a structure decorated by a new name. Therefore such expressions are called GENERATIVE STRUCTURE EXPRESSIONS.

Each elaboration of a data type declaration (`datatype` ...) yields a fresh type i.e., a type decorated by a new name.

```
structure Stack_n1 =
struct
 type elt_int = int
 datatype stack_t1 = ST of elt list ref
 val initStack_t1 = ST(ref[ ])
end

structure StackUser1_n2 =
struct
 structure Stack1_n1 = Stack
 ...
end

structure StackUser2_n38 =
struct
 structure Stack2_n1 = Stack
 ...
 datatype stack_t23 = ST of elt list ref
end
```

To be complete, one would have to decorate each structure not merely by a name but also with its decorated components and subcomponents. (A structure expression in the form of a functor application does not in itself reveal the components of the resulting structure.) However, to keep decorations to a minimum, we shall usually not spell out the decorated subcomponents.

3.4 Decorating Signatures

```
signature StackSig(m1,s1,s2) =
sig_m1
 type elt_s1
 type stack_s2
 val new_unit→s2 : unit->stack
end

signature TranspSig(m1,s1) =
sig_m1
 type elt_s1
 type stack_t1
  sharing type stack_t1 = Stack.stack_t1
 val new_unit→t1 : unit->stack
end
```

Bound names are collected at the signature identifier. They are listed between parenthesis to indicate that they are merely place holders.

The bound names of *StackSig* are $m1$, $s1$ and $s2$. The free names of *StackSig* are *unit* and $\rightarrow$. The bound names of *TranspSig* are $m1$ and $s1$. The free names of *TranspSig* are $t1$, *unit* and $\rightarrow$.

Exercise 16 Consider

```
signature Symbol =
sig
 type symbol
 type value
 sharing type value = int
end
```

Decorate this signature declaration with type and structure names. How many bound names are there? How many free?

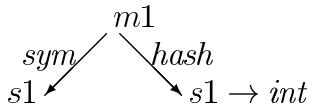
If two structures are found to share by the static analysis then they really are the same at run-time. Therefore, when decorating signatures one must make sure that if two structures are made to share (by being given the same name) then any type or structure which is visible in both structures must be made to share as well.

Exercise 17 Consider the following signatures most of which you have already seen in Lecture 1.

```
signature ValSig =
sig
 type value
end

signature SymSig =
sig
 eqtype sym
 val hash : sym->int
end

signature LexSig =
sig
 structure Sym : SymSig
 val getsym : unit->Sym.sym
end

signature SymTblSig =
sig
 structure Val: ValSig
 structure Sym: SymSig
 type table
 val lookup:
     table * Sym.sym-> Val.value
 ...
end

signature ParseSig =
sig
 structure Lex : LexSig
 structure Tbl : SymTblSig
  sharing Lex.Sym = Tbl.Sym
```

```
 type abstsyn
 val parse : unit->abstsyn
end
```

Decorate these signatures. When one signature refers to another (for instance *LexSig* refers to *SymSig*) you should put a full decoration on the structure identifier (*Sym*), i.e. a decoration which shows both a name and the subcomponents of the structure. Full decorations can be drawn as trees; in the example at hand you can decorate *Sym* by



Make sure that you decorate shared substructures (for instance *Sym* in *ParseSig*) consistently so as to represent that sharing of two structures implies sharing of their substructures.

3.5 Signature Instantiation

```
structure Stack_n1 =
struct
 type elt_int = int
 datatype stack_t1 = ST of elt list ref
 fun new_unit→t1() = ST(ref[ ])
end

signature StackSigA_(m1,s1,s2) =
sig_m1
 type elt_s1
 datatype stack_s2 = ST of elt list ref
 val new_unit→s2: unit->stack
end
```

Note that if we substitute $n1$ for $m1$, *int* for $s1$ and $t1$ for $s2$ in the decoration of *StackSigA* then we get the decoaration of *Stack*. We say that *Stack* is an **instance** of *StackSig*. More generally, we say that a structure is an INSTANCE of a signature if the decoration of the former is obtained from the decoration of the latter by performing a substitution of names for the **bound** names of the signature (the free names of the signature must be left unchanged). The process of substituting names for bound names is called REALISATION.

```
structure Stack_n1 =
struct
 type elt_int = int
 datatype stack_t1 = ST of elt list ref
 fun new_unit->t1() = ST(ref[ ])
end

signature StackSigB_(m1,s1) =
sig_m1
 type elt_s1
 datatype stack_t1 = ST of elt list ref
 sharing type stack_t1 = Stack .stack_t1
 val new_unit->t1: unit->stack
end
```

Stack is an instance of *StackSigB* via the realisation $\{m1 \mapsto n1, s1 \mapsto int\}$.

```
structure OddStr_n1 =
struct
 type elt_int = int
 val test_bool = false
end

signature WrongSig_(m1,s1) =
sig_m1
 type elt_s1
 val test_s1 : elt
end
```

OddStr is not an instance of *WrongSig*, for $s1$ would have to be realised by *int* (because of *elt*) but then *test* is decorated by *int* in the signature and by *bool* in the structure.

3.6 Signature Matching

Matching of a structure against a signature is a combination of two operations. The first, signature instantiation (described above), is concerned with instantiating the bound names of the signature to the "real" names of the structure. The second is concerned with ignoring information in the structure which is not required by the signature.

```
structure Tree_n1 =
struct
 datatype 'a tree_t1 = LEAF of 'a
                     | NODE of 'a tree * 'a tree
 type intTree_int t1 = int tree
 fun max(a:int, b:int) =
  if a > b then a else b
 fun depth_a t1->int(LEAF _) = 1
 |   depth(NODE(left,right))=
       max(depth left, depth right)
 end

signature TreeSig_(m1,s1,s2) =
sig_m1
 type 'a tree_s1
 type intTree_s2
 fun depth_s2->int: intTree->int
end
```

A structure MATCHES a signature if the structure can be cut down to an instance of the signature by

1. forgetting components;
2. forgetting polymorphism of variables.

Tree matches *TreeSig*. First perform the realisation $\{m1 \mapsto n1, s1 \mapsto t1, s2 \mapsto int\,t1\}$ on the signature. The resulting decoration can be obtained from the decoration of *Tree* by

1. forgetting the constructors *LEAF* and *NODE*
2. instantiating $'a\,t1 \to int$ to $int\,t1 \to int$ (i.e. the realisation of $s2 \to int$)

Exercise 18 Let *mytype* be a type which is declared in a structure and specified in a signature. In which of the following cases can the structure match the signature?

(1) *mytype* is declared as a `datatype` and specified as a `datatype`.

(2) *mytype* is declared as a `datatype` and specified as a `type`;

(3) *mytype* is declared as a `type` and specified as a `type`;

(4) *mytype* is declared as a `type` and specified as a `datatype`.

3.7 Signature Constraints

```
structure Tree: TreeSig =
 struct ... end
```

In a structure declaration with an explicit signature constraint, the resulting view of the declared structure is precisely the one given by the instantiated signature.

In the example above, the resulting view of *Tree* will hide the constructors *NODE* and *LEAF* and the function *max*. Notice, however, that *intTree* is decorated by the instance of $s2$, i.e. by *int* $t1$, where $t1$ is the decoration of *'a tree*. Consequently, *Tree.intTree* and *int Tree.tree* now mean the same thing, namely *int* $t1$. This sharing was obtained through relisation — it was not explicit in *TreeSig*.

In short, explicit signature constraints can remove components and polymorphism, but they do not affect existing sharing.

Here is an example of a structure declared with an explicit signature constraint. You should convince youself that the structure really does match the signature.

```
signature SymSig =
sig
 type sym
 type code
  sharing type code = int
 val hash : sym->int
 val mksym : string->sym
 val nameof : sym->string
end

structure Sym : SymSig =
struct
 datatype sym = SYM of string * int
 type code = int
 fun convert(s: string): code = ...
 fun hash(SYM(s, n))= n
 fun mksym(s) = SYM(s, convert s)
 fun nameof(SYM(s,_))= s
end
```

Exercise 19 Complete the declaration of *convert*.

Exercise 20 Declare a different structure *NewSym*, also constrained by *SymSig*, such that *NewSym.sym* shares with *string*. Which of the following expressions are valid?

(1) `"a" ^` *NewSym.mksym* `"d"`

(2) `"a" ^` *Sym.mksym* `"d"`