Attribute systems and plumbing diagrams

Andrew Seniuk Robin Cockett

March 26, 2007

1 Introduction

The purpose of these notes is to provide a gentle introduction to attribute systems using plumbing diagrams. These notes were developed to supplement the course material of CPSC411, Introduction to Compilers, and employ a graphical representation of attribute systems, called "plumbing diagrams", in order to represent, particularly, the semantic analysis stage of a simple compiler. Attribute systems are usually associated with a context free grammar, however, in these notes and in the class we associate them more generally with datatypes and view them as a way of expressing a computation on a datatype. This makes them more generally applicable and provides those who get to understand what they are about a powerful program development tool.

2 What is an attribute system?

From a pragmatic viewpoint, an attribute system is a specification of a program. Given a datatype, an attribute is some computable characteristic (*e.g.* numeric value, type, or space requirement) for which a program is wanted. An attribute system indicates how the attribute is calculated, and usually this involves showing how it is computed from other attributes. A given datatype may have many different attribute systems associated with it, each one for computing corresponding attributes. Once you have a valid attribute system it can be used as a template to write actual code which, if the translation is done correctly, is guaranteed to work.

Attribute systems at first sight are rather complex things so we shall approach them from an informal point of view to start with. Plumbing diagrams such as the sample shown in Figure 1 provide a visual representation of the complexities of attribute systems and will be used throughout these notes. When developing your own attribute systems, you are strongly encouraged to work them out (and present them) as plumbing diagrams.

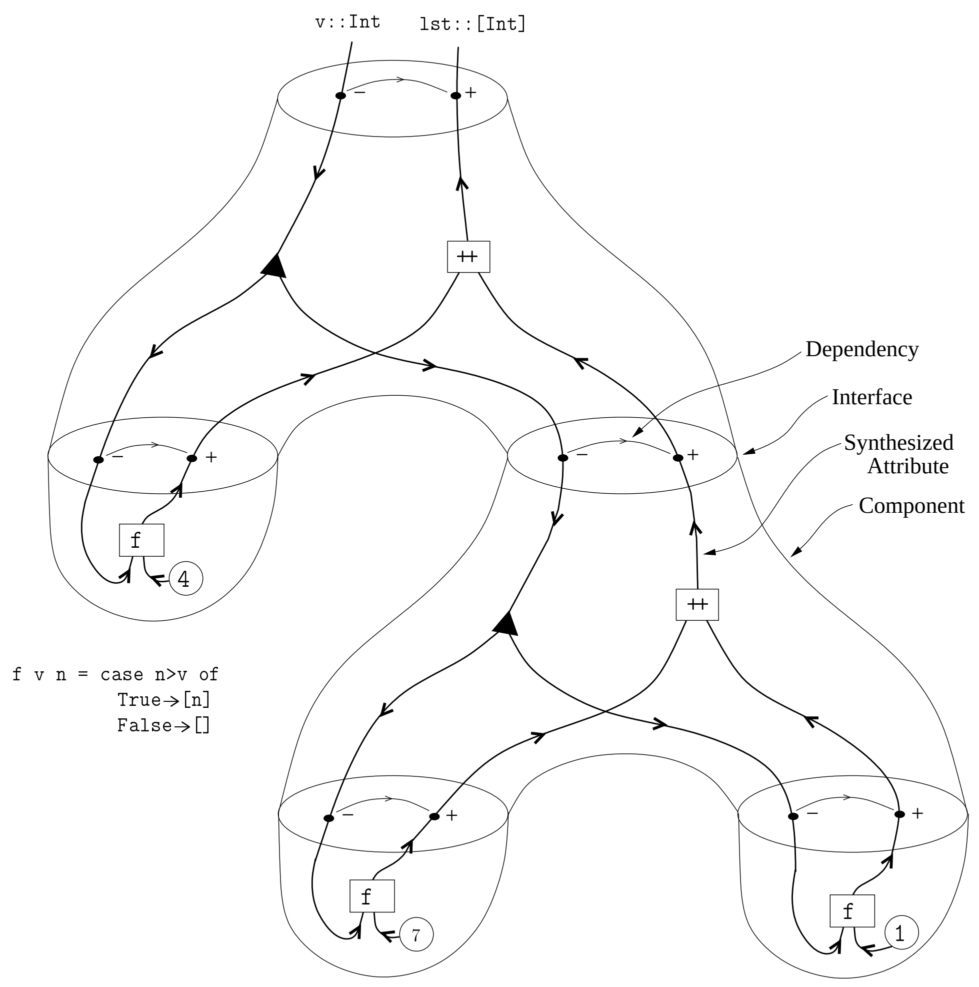


Figure 1: Typical plumbing diagram of an attribute system.

An attribute system is made up of components, which may be connected together in any number and arrangement consistent with the types of the interfaces. One may think of each component as a piece of plumbing and the interface as a plug: each plug has a special configuration and two components can only be plugged together when the plugs at that interface match. The components therefore naturally form a datatype in which the components are constructors whose types determine the interfaces. In the example of Figure 1, the datatype is simply a binary tree of integers:

```
Tree Int = NODE (Tree Int) (Tree Int)
         | LEAF Int
```

This datatype has two constructors, and so there are two kinds of components possible in attribute systems for this datatype, one for `NODE` and one for `LEAF`. In our particular example, the attribute system computes the list of integers from the leaves of the tree (in left to right order) which have value greater than a threshold `v`.

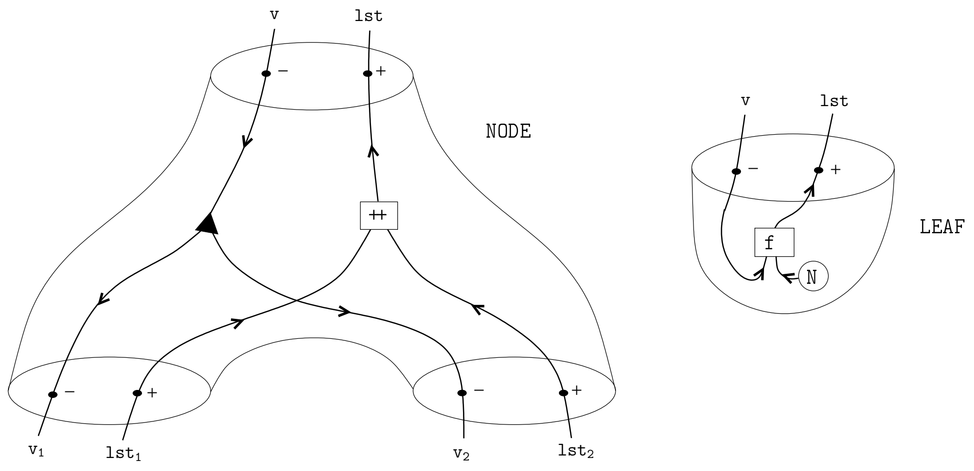


Figure 2: The two kinds of components possible in our attribute system of Figure 1.

The `NODE` component performs two computations (also called equations in the textbook):

- The integer `v` is copied into $\mathtt{v}_1$ and $\mathtt{v}_2$.
- Lists $\mathtt{lst}_1$ and $\mathtt{lst}_2$ are concatenated to give `lst`.

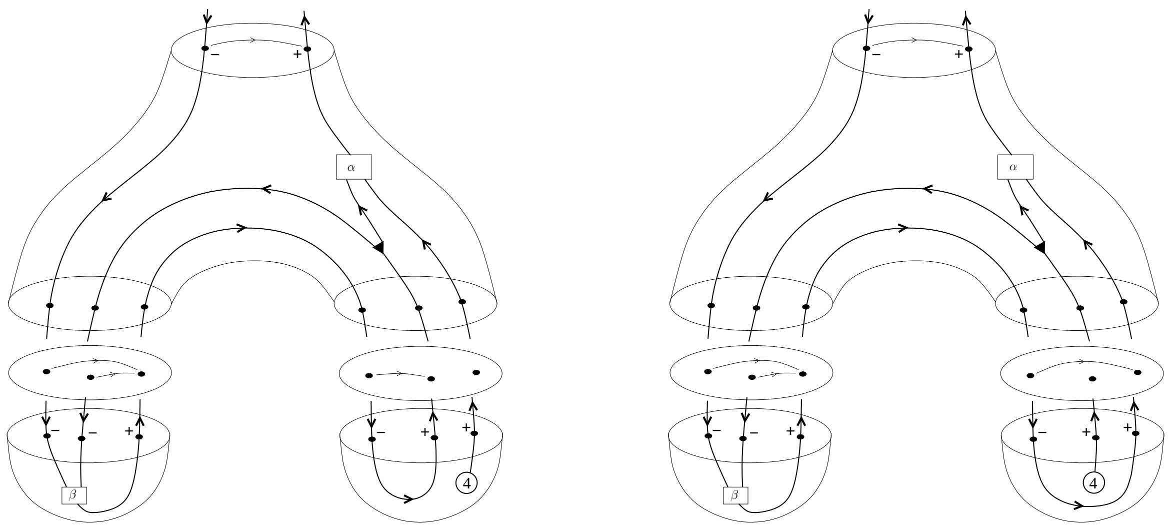
The `LEAF` component performs one computation: It applies the function `f` to the inherited attribute `v` and the constant integer `N`, which returns a list either empty or singleton.

```
f v n = case n>v of
          True→[n]
          False→[]
```

Bear in mind that, while the computation in this example is very basic, it's the idea of an attribute system which is the point. Attribute systems provide us with a powerful tool for organizing and writing correct code in cases where the computations involved are monstrously complex.

However, there is a danger of things going awry. Even though each component represents a sensible computation, if precautions aren't taken it may happen that a connected assemblage represents a nonsensical computation. This is illustrated in Figure 3, to which we'll return for a thorough analysis in Section 5.

Since there is no limit to the number of components which can be assembled, and problems might only become apparent in some very large assemblies, checking the validity of an attribute



Plumbing diagram exhibiting circular dependencies. This one represents a valid computation.

Figure 3: Depending on how components are connected up, cycles of dependencies may be possible; component definitions which can give rise to such constructions do not constitute a valid attribute system.

system cannot be reduced to checking all possibilities. This is where some theory becomes useful: in particular conditions regarding *circularity* will come to the rescue, as we'll see after defining attribute systems more formally.

3 Formal definition of an Attribute System

Here is the definition of an attribute system that we shall use. This is sometimes called a "strongly non-circular" attribute system in compiler texts.

An **attribute system** consists of:

1. A mutually recursive collection of inductive datatypes (or a context free grammar).
2. For each type (non-terminal) a set of attributes which are labelled as either inherited ($-$) or synthesized ($+$).
3. For each type (non-terminal) a dependency specification for each synthesized attribute upon the inherited attributes.
4. With each constructor (production), a function for calculating:
   (a) each synthesized parent attribute, and

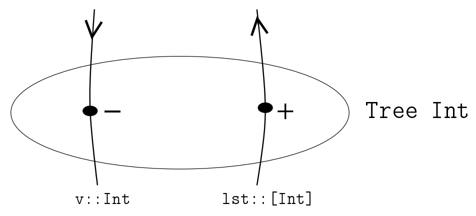


Figure 4: Synthesized and inherited attributes

   (b) each child attribute to be inherited by underlying components.

5. The condition of strong non-circularity must be satisfied by the data above:
   (a) When the dependencies of the children types (RHS nonterminals) are added to the dependencies given by the functions, the dependency graph must have no cycles.
   (b) At the parent interface (LHS nonterminal), the dependencies implied by this graph between the attributes must all lie within the specified dependencies.

Given such a strong attribute system it is possible to automatically generate code which will calculate the attributes. Let's take a closer look at each of these defining properties.

1. A mutually recursive collection of inductive datatypes. We gave an example having one datatype with two constructors; typical applications (such as intermediate code generation) are much more complex than this example! We'll look at some of these in later sections.

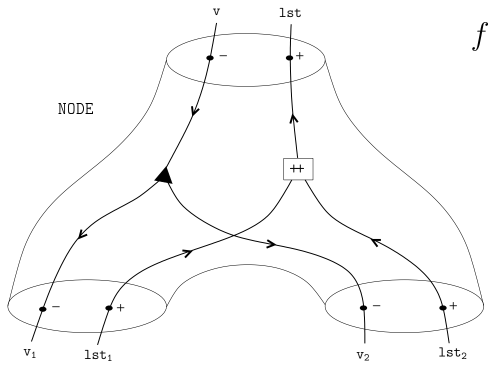
```
Tree Int = NODE (Tree Int) (Tree Int)
         | LEAF Int
```

2. For each type a list of inherited and synthesized attributes. (These are the same for all constructors of the type.)

3. For each type a list of dependencies. For each synthesized attribute, one possible dependency may exist from each inherited attribute. In this example, there is only one datatype, and only one inherited and one synthesized attribute, so the list of dependencies is simply `v` $\vdash$ `lst`. Dependencies are indicated in plumbing diagrams by small directed arcs between the attributes at an interface, as illustrated in Figure 1.

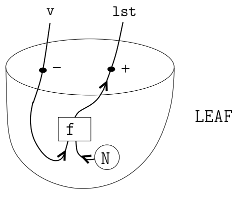
4. A description of how you calculate the synthesized attributes in the head of a component from inherited attributes from the parents and synthesized attributes from the children. These descriptions are normally in the form of equations or functions. In this document some of these are presented in a Prolog-like "definite clause language" (DCL) specifically designed for attribute

systems. Others are presented in Haskell. Our present example introduces the DCL calculations for `NODE` and `LEAF`.



$$\begin{aligned} & f[NODE(t_1,t_2)](x;lst) \\ & \quad :- \quad f[t_1](x;lst_1)\,, \\ & \qquad\quad\; f[t_2](x;lst_2)\,, \\ & \qquad\quad\; lst := lst_1 {+\!\!+} lst_2\,. \end{aligned}$$

$$\begin{aligned} & f[LEAF(n)](v;lst) \\ & \quad :- \quad v < n\,, \\ & \qquad\quad\; lst := [n]\,, \\ & \quad :- \quad lst := []\,. \end{aligned}$$



Probably the semantics of this DCL code are quite clear from a comparison with Figures 1 and 2. The idea of a DCL clause is that it acts on a datatype(s) (in this case trees). For each clause what is in square brackets is matched to the datatype(s) being processed. The clause then has arguments: those before the semicolon are input (inherited) those after are outputs (synthesized). The body of the clause after ":-" and before the "." are statements indicating what should be done. If a statement in the body of a clause is a condition (*e.g.* $x < n$) which fails then the statements of the clause are skipped till a handler is found (here the phrase starting with ":-"), otherwise one continues. Thus the second clause calls for n to be returned as a singleton list only if it is greater than x.

5. The conditions of an attribute system must be satisfied

(a) No cycles occur in the directed graph consisting of circuitry inside a component together with the terminating (bottom, RHS) dependencies.

(b) The dependencies implied on the attributes in the head (top, LHS) of a component by this digraph are contained in specified dependencies. A synthesized attribute is dependent on an inherited attribute when to compute that attribute one may need to use the values of the inherited attribute: this means there should be a computation path (in the circuitry) from the inherited attribute to the synthesized attribute.

4 Plumbing Diagrams

Plumbing diagrams are a notation – a visual language – for attribute systems and their instances. As seen in Figure 2 and Figure 1, respectively, they are an intuitive component-based way to visually sum up all the elements of an attribute system and to present an arbitrary instance. A component's surface (the "pants") represents the datatype constructor declaration (*e.g.* `Tree Int = NODE (Tree Int) (Tree Int)`). The interface (aperture) at the top of a component represents the parent type of the constructor, and is always present. A component also has one bottom (child) interface per argument to the data constructor. The attributes are represented as wires passing in or out of parent and child interfaces. Interfaces are labelled by their types, and can be plugged together only if they are of identical types.[1] The functions relating attributes are represented as intra-component circuitry and multi-input/multi-output function boxes. Two types of functions have distinguished notation: small, filled black triangles represent single-input/many-output fanout (copy), and small, non-filled circles or ovals represent terminal values constructed by the datatype. These latter elements behave like constant function boxes in the plumbing diagrams, and represent actual data embedded in the datatype. For function boxes, the actual code must be given somewhere – except when it is standard (such as `++`). Note there is no requirement for function box inputs or outputs to pass either up or down; and indeed any input or output line might connect to the parent interface, a ch ild interface, or another function box, provided that the conditions of an attri bute system are inviolate.

There is a one-to-one correspondence between kinds of plumbing components and constructors of datatypes, although when a constructor takes a primitive type argument (such as `Int`), this value is shown in a circle within the component rather than creating separate components for each possible value of the primitive type. From the perspective of a context tree grammar (rather than the datatype), these are the terminals. It is here that the datatype (or grammar) has a direct influence on the attributes beyond a generic tree-structural one. We see this in the example of Figure 1 with the integer constants. Note that an attribute computation may involve constants of types which are quite independent of the datatypes of the attribute system. In such a case, the constant is represented as a function box with no inputs – circles are reserved for constants obtained from the datatype instance itself.

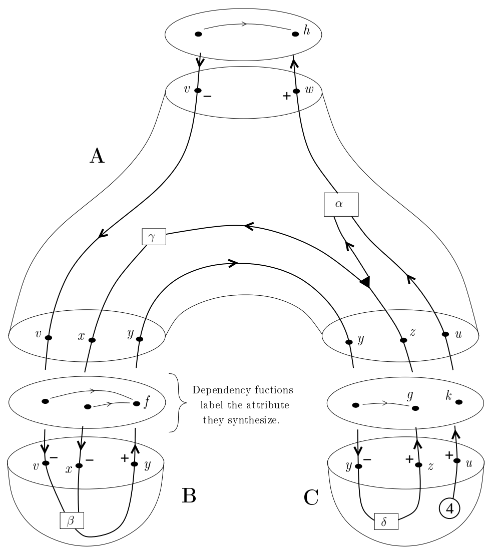
Finally, one should bear in mind the distinction between the plumbing diagrams for the components of an attribute system (Figure 2), and the plumbing diagram of an instance of the attribute system (Figure 1) in which not all components need appear but they are connected into a configuration with only one root interface free. A contemplation of the range of structures entailed by this model should convince you that it is very expressive. The terminal data circles in a plumbing diagram of an abstract component can be thought of as empty, typed slots. The same circles in the plumbing diagram of a concrete instance of the (strict) datatype would contain type-compatible values in these slots.

Some parallels that exist between plumbing diagrams, attribute systems, context free grammars and datatypes are summarized in Table 1.

[1]It should go without saying that no gross cycles are allowed by plugging any child interface into the parent interface of any ancestor! This ruled out in our definition of an attribute system because the ductwork is supposed to represent a instance of an inductive datatype (or parse tree of a c ontext free grammar), which is always acyclic.

Table 1: Some correspondences

Plumbing	Attribute System	Datatype
Ductwork	Production	Constructor
Interface	Nonterminal	Types
Inherited and synthesized attributes		Type of (higher-order) fold
Circuitry	Semantic rule	Function used in fold



$$\begin{array}{ll}
h[A(b,c)](v;w) & \\
\quad \text{:-} & f[B](v,x;y), \\
& g[C](y;z), \\
& k[C](;u), \\
& w := \alpha(z,u), \\
& x := \gamma(z). \\
f[B](v,x;y) & \\
\quad \text{:-} & y := \beta(v,x). \\
g[C](y;z) & \\
\quad \text{:-} & z := \delta(y). \\
k[C](;u) & \\
\quad \text{:-} & u := 4.
\end{array}$$

Figure 5: Tracing out the pathology of a circularity.

5 Strong Non-Circularity

Given an arbitrary assemblage of components with their attributes and circuitry[2] (*vis.* a plumbing diagram), there is no guarantee that the attempt to calculate an attribute will not lead to an infinite (circular) calculation. Let's consider the example of Figure 3(a) again: Figure 5 gives the offending instance, with copious labelling. We've added a few function boxes to make the example less perfunctory. Let's try to write some code for the three constructors involved here and see what happens.

Clearly there is no way to write a finite computation of attribute w in this instance, as w depends on z, z depends on y, y depends on x, and x depends on z.[3] So, it is not enough that the attributes at the interfaces of the components all match up. What you need to ensure is that, given any

[2]Notice we don't call it an attribute system, since then by definition it would not misbehave.

[3]Indeed, even if w had no dependence on the attributes x, y or z, this component would still violate 5(a) in our definition of an attribute system.

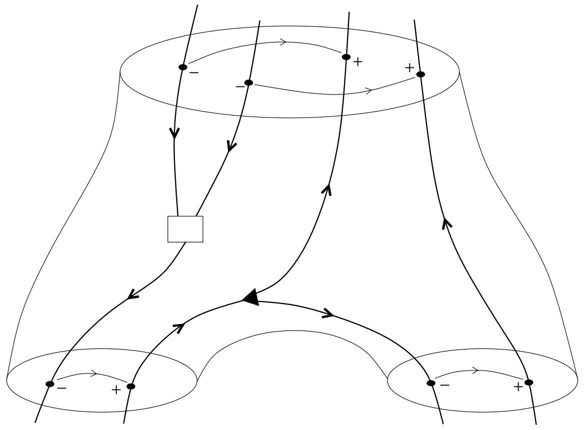


Figure 6: Violation of condition 5(b) for strong non-circularity: There are two implied dependencies on the parent interface which are not part of the specified dependencies, presuming that all specified dependencies are shown.

instance of the datatype (parse tree) and any attribute, it is always possible to calculate it. This means that whatever tree you build, the calculations associated with the attributes must **never** involve a cycle. Given that there may be infinitely many possible such trees you should note that this is not something that can be simply checked by examining the possibilities!

It is possible to determine whether a "general" attribute system, as above, is non-circular but the algorithm is exponential.[4] However, there is a stronger notion, often called *strong non-circularity*, which is easy to check and is, in fact, the condition which is used in practise. The two requirements for strong non-circularity were given by 5(a) and 5(b) in the definition of an attribute system. 5(a) prevents explicit circular calculations within a component, and 5(b) checks that the specified dependencies include all the implied dependencies which result from the circuitry. A failure of 5(a) indicates an error in your logical plan! Failure of 5(b) is more complicated: it may mean you need to add some more specified dependencies to your attribute system (*i.e.* your choices in step 3 of the definition were not sufficiently complete), after which 5(a) needs to be checked again and, considering that the graph will now have additional arrows, the risk of a failure of condition 5(a) will increased. The nice thing about strong non-circularity is that one can check it merely be running these checks on each component in isolation. However, there do exist computable attribute systems which violate strong non-circularity – we are only assured that every strongly non-circular attribute system is computable, not that such systems describe every possible computation.

Given the specified dependencies on the children nonterminals (RHS or argument types) and the dependencies implied by the equations it is possible to generate for each production (or constructor) a little directed graph where an arrow indicates a dependency. If this graph has no cycles *and* the implied dependencies on the parent are always contained within the specified dependencies for

[4]See the text: in fact this check relies on determining all the possible dependencies which is what makes it exponential.

that nonterminal (or type) then the attribute system will be non-circular.

The proof of this is as follows: suppose there is a (parse) tree with a cycle of dependencies then there is a topmost nonterminal (type) through which this cycle passes. The cycle below this nonterminal (type) produces a dependency $-N.a \rightarrow +N.b$. This must be one of the specified dependencies (a simple inductive argument on the underlying tree). But the cycle is formed by a dependency $+N.b \rightarrow -N.a$ above the nonterminal which produces an instantaneous cycle in the calculation of the graph for that production (constructor).

This means you can write the code!

6 Applications

6.1 Code to check that numbers are well-formed

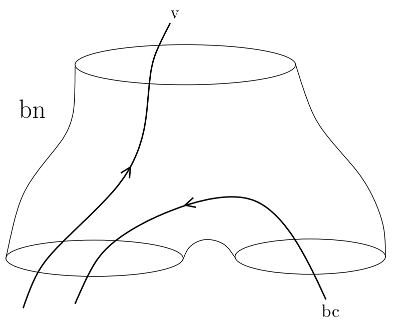
As a one-paragraph crash course in the definite clause language (DCL), consider some example code

```
base_num [bn(N,B)](;z)          dig [n](b;m)
  :- base[B](;b),                 :- n < b,
     num[N](b;z).                    m := n.
```

Each group of statements corresponding to a constructor is a 'clause'. The phrases of each clause can contain guards which must evaluate to true in order to continue the normal evaluation of clause. If a guard fails one looks for an exception handler. In Haskell this could be modelled with an exception monad like `Maybe` or `Error` from the Haskell Standard Prelude. For example, `dig` $[n](b;n) = n < b$. fails *unless* $n < b$ in which case it synthesizes n.

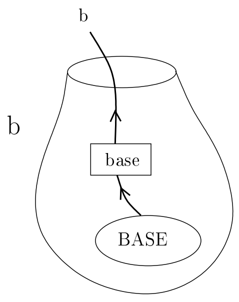
We're now ready to tackle the complete problem. Consider the following grammar for non-negative integers, of base either octal or decimal. The plumbing diagram in Figure 7 illustrates a possible parse relative to this grammar, along with attribute calculations for the value of the number. Note that this value is a magnitude and has no particular base associated with it – the base is a feature of the representation of the number, not a feature of its value.

```
(bn)    based_num →  num basechar.
(bo)    base      →  OCTAL
(bd)              |  DECIMAL.
(num1)  num       →  num digit
(num2)            |  digit.
(dig)   digit     →  0 | 1 | 2 | ...  | 9.
```



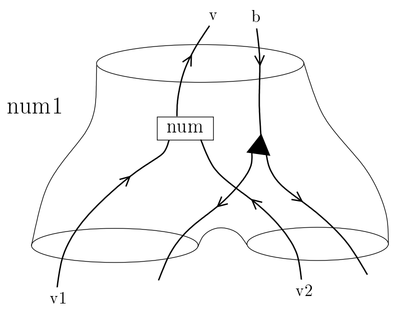
based_num[bn(N, B)]($; z$)
  :- base[B]($; b$),
     num[N]($b; z$).

```
based_num (BasedNum n bc) = z
  where b = base bc
        z = num n b
```



base[OCTAL]($; 8$).
base[DECIMAL]($; 10$).

```
base OCTAL = 8
base DECIMAL = 10
```



num[num1(N, D)]($b; v_1$*b+v_0)
  :- num[N]($b; v_1$),
     dig[D]($b; v_0$).

```
num (Num1 n d) b = testOkay [v1,v0] v
  where v1 = num n b
        v0 = dig d b
        v = ((fromOkay v1)*b+(fromOkay v0))
```

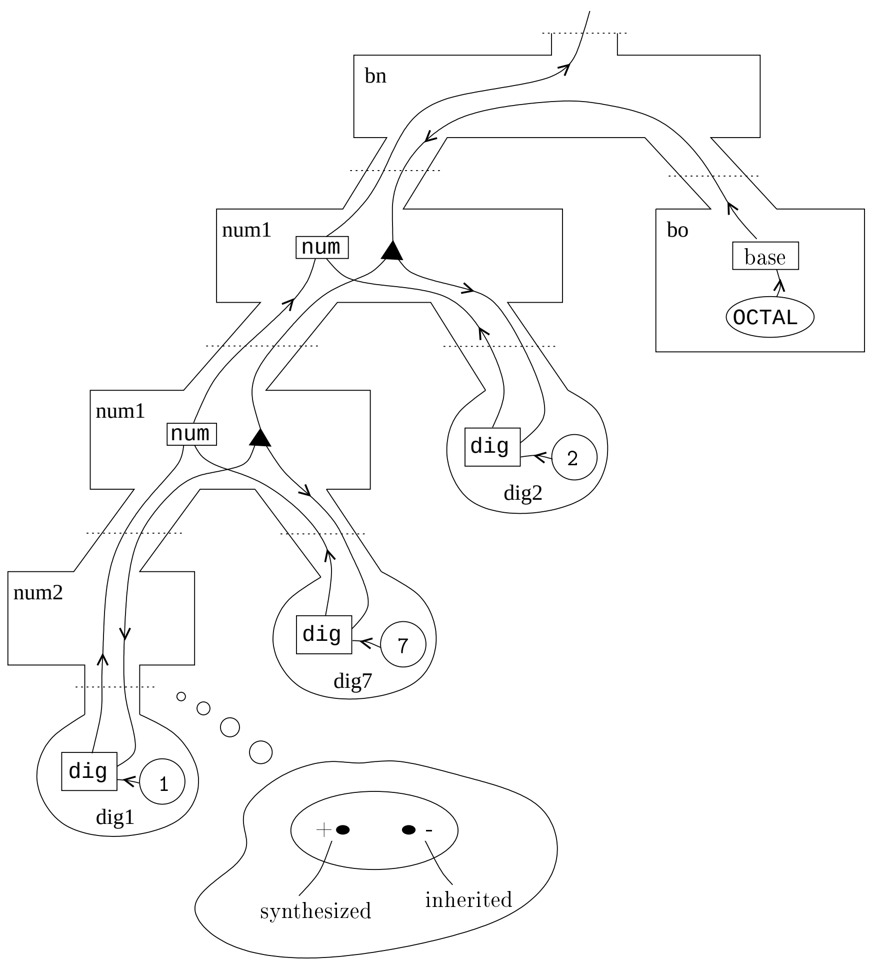
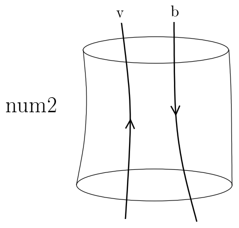
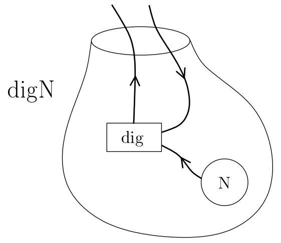


Figure 7: Plumbing diagram for computing the magnitude of the number encoded as "172o"



$\texttt{num}[\texttt{num2}(D)](b; v)$
  $\texttt{:-}\ \texttt{dig}[D](b; v)$.

```
num (Num2 d) b = v
  where v = dig d b
```



$\texttt{dig}[n](b; n)\ \ \texttt{:-}\ \ n < b$.

```
dig n b
  | n < b = Okay n
  | otherwise = NotOkay
```

The code for this computation is expressed in both DCL and Haskell. For more information about the `Okay` monad, you can refer to Appendix A, but roughly speaking, `testOkay` returns its second argument (wrapped in an `Okay`) if every element of the list in the first argument is `Okay`, and returns `NotOkay` otherwise.

6.2 A simple symbol table, and `let` expressions

The symbol table is a list of identifiers $[x_1, y_1, x_2, ...]$ supporting two operations: insertion, and membership test.

insert(x,st;st') to add an element to the front of the list.

member(x,st;) to check for membership in the list.

Consider the following simple expression grammar.[5]

```
exp    →  exp + exp
        | (exp)
        | ID
        | NUM
        | LET decls IN exp.
decls  →  decls , dec
        | dec.
dec    →  ID = exp.
```

Grammar

```
data exp   =  ADD(exp,exp)
           |  PAR(exp)
           |  ID(string)
           |  NUM(int)
           |  LET(decls,exp)
data decls =  MANY(decls,dec)
           |  ONE(dec)
data dec   =  DEC(string,exp)
```

Syntax Tree

A typical expression is then

```
let y = 22 in
 ( let x = 4+y , z=7+y in x+z+y )
```

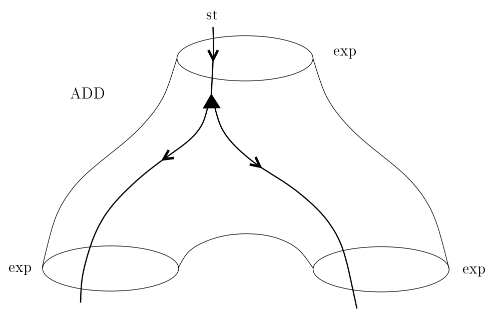
Given an expression and a symbol table, how does one check whether the expression is well-formed? It is well-formed if the values of variables are all declared before they are used. Thus the above is well formed. Here is a summary of the scoping rules:

- The usual scope rule: that is if v is declared in D then v can be used in the term t in `let D in t`).

[5]This grammar is both left-recursive and ambiguous! However, we shall just use the parse trees as the basis for an attribute grammar; the technicalities of the parse are immaterial here.

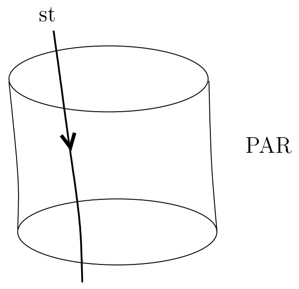
- Simultaneous declaration semantics: a declaration can only use variable which are already declared and these do not include the variables in the same declaration list. Thus, `let x=3, y=x in 2y` is not legal.

In this attribute system if a retrieveal from the sytmbol table (which is a membership test) fails, the whole expression is immediately deemed to be ill-formed. The following are the components of an attribute system implementing this solution, with the plumbing diagram for an example expression given in Figure 6.2. For the last three, the first symbol table is for checking expressions, and the second symbol table is for collecting declarations.



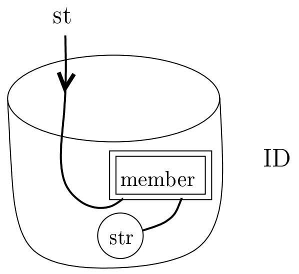
valid_exp[ADD(t_1, t_2)](st;)
:- valid_exp[t_1](st;),
valid_exp[t_2](st;).

```
valid_exp (ADD t1 t2) st =
  valid_exp t1 st
  valid_exp t2 st
```



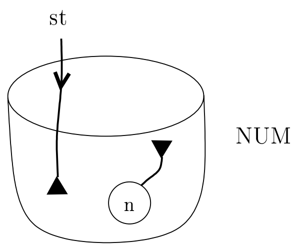
valid_exp[PAR(t)](st;)
:- valid_exp[t](st;).

```
valid_exp (PAR t) st =
  valid_exp t st
```



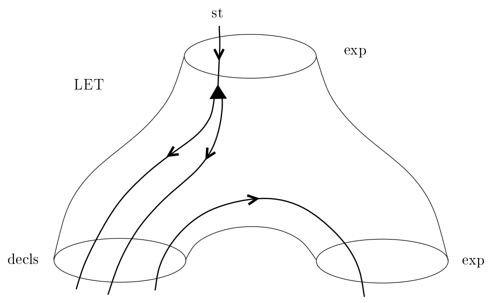
valid_exp[ID(str)](st;)
:- member(str,st).

```
valid_exp (ID str) st =
  member str st
```



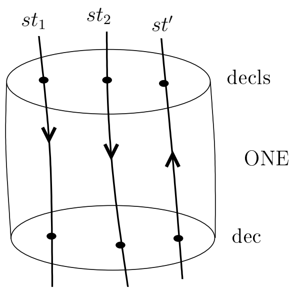
valid_exp[NUM(n)](st;).

```
valid_exp (NUM n) st = True
```



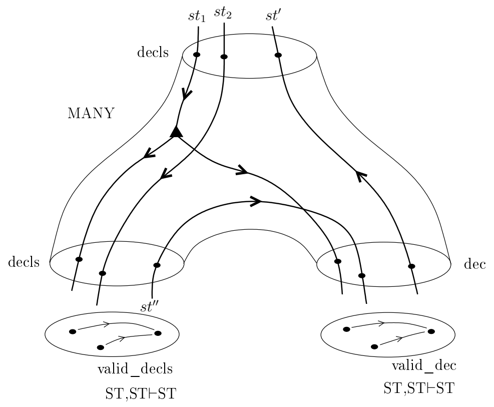
valid_exp[LET(decls,exp)](st;)
:- valid_decls[decls](st,st;st')
valid_exp[exp](st';).

```
valid_exp (LET decls exp) st
 = valid_exp exp st'
 where st' = valid_decls decls st st
```



valid_decls[ONE(dec)]$(st_1, st_2; st')$
:- valid_dec[dec]$(st_1, st_2; st')$.

```
valid_decls (ONE dec) st1 st2 = st'
 where st' = valid_dec dec st1 st2
```



valid_decls[MANY(decls,dec)]$(st_1, st_2; st')$
:- valid_decls[decls]$(st_1, st_2; st'')$,
valid_dec[dec]$(st_1, st''; st')$.

```
valid_decls (MANY decls dec) st1 st2 = st'
 where st'' = valid_decls t st1 st2
       st' = valid_dec dec st1 st''
```

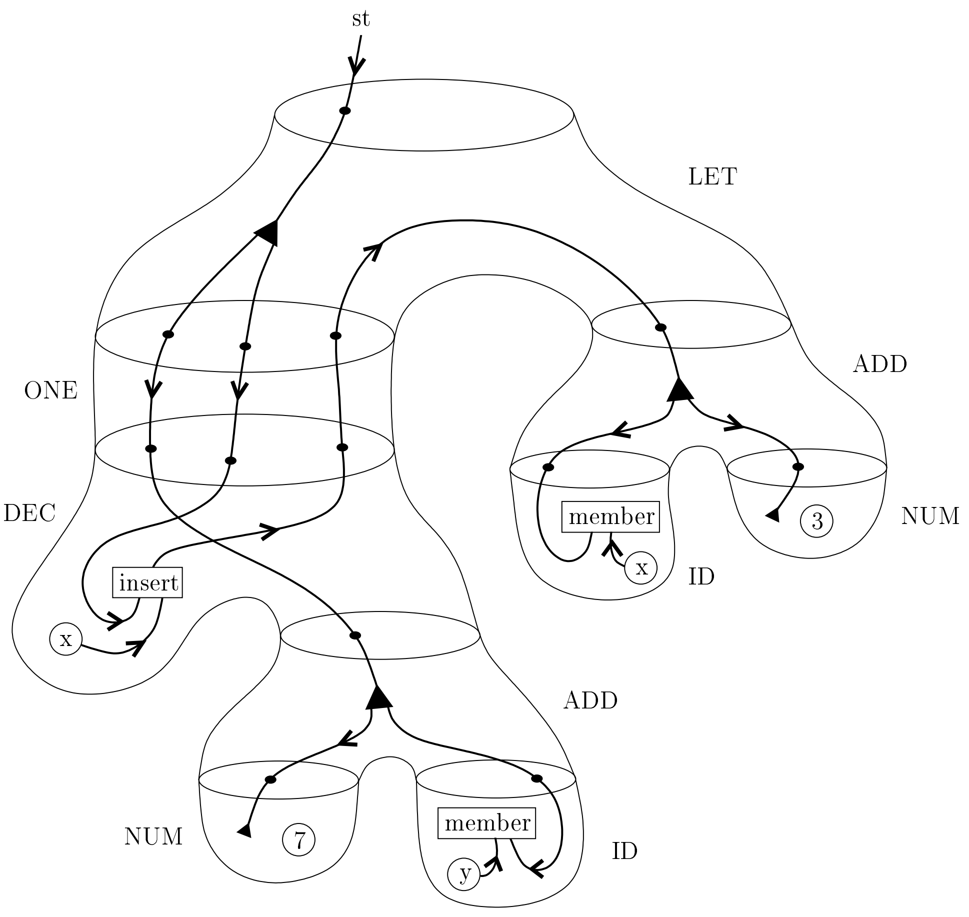
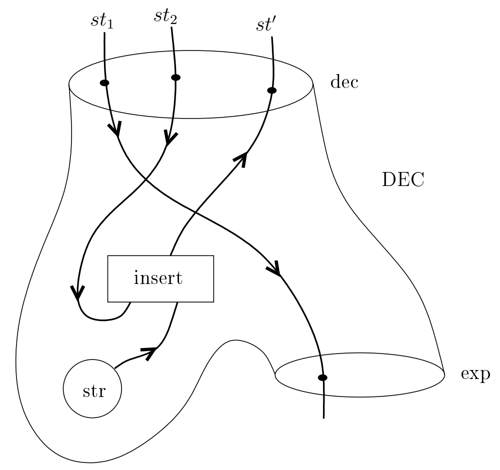
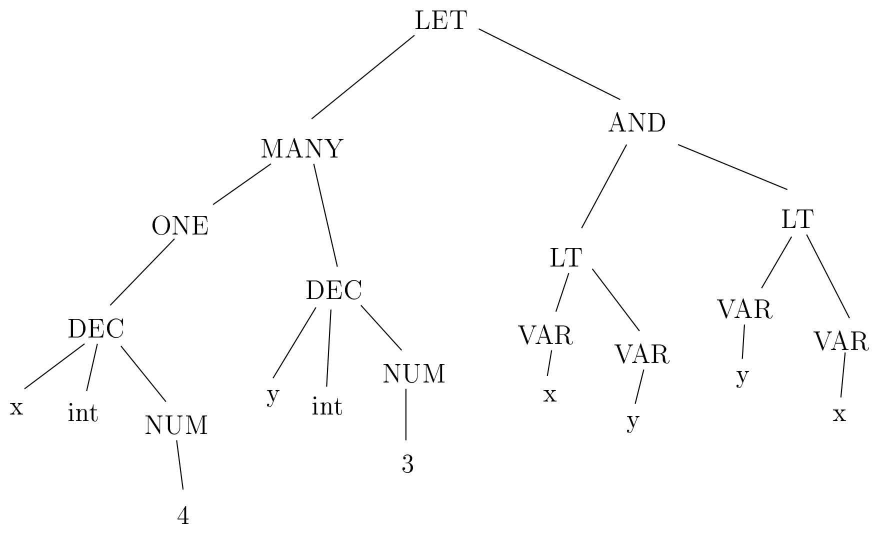


Figure 8: Plumbing diagram for testing validity of `let x=7+y in 3+x`. The outcome would depend on whether the incoming table satisfied `member(y,st;)` or not.



$$\begin{array}{l}\text{valid\_dec[DEC(str,exp)]}(st_1, st_2; st') \\ \quad \text{:- insert}(\mathit{str}, st_2; st'), \\ \quad\quad \text{valid\_exp[exp]}(st_1;).\end{array}$$

```
valid_dec (DEC str exp) st1 st2
  | valid_exp exp st1 = insert str st2
  | otherwise = error "valid_dec: error"
```



LT : int,int → bool

AND : bool,bool → bool

ADD : int,int → int

Figure 9: Does `let x = 4, y = 3 in (x<y) & (y<x)` type check?

6.3 Simple type checking

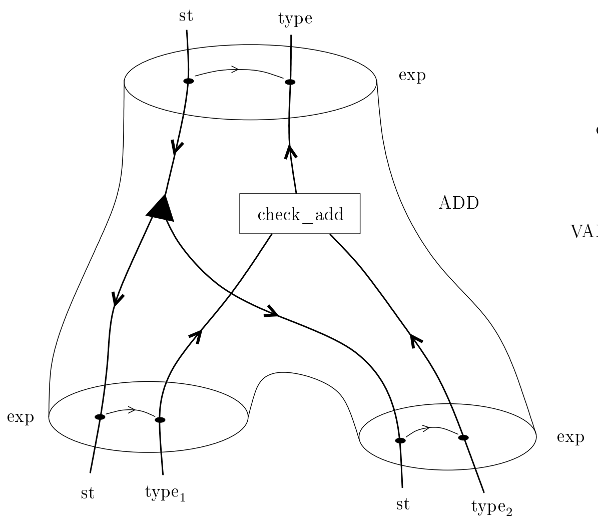
Now let's see how we can expand on this, and – augmenting the grammar slightly – modify our attribute system to perform type checking on expressions.

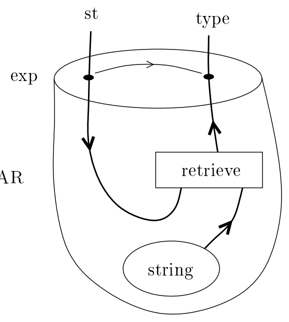
```
exp    →  exp + exp  |  exp & exp
       |  exp < exp  |  (exp)
       |  ID  |  NUM
       |  LET decls IN exp.
decls  →  decls , decl
       |  decl.
decl   →  ID = exp.
```

```
data Exp    =  ADD Exp Exp   |  AND Exp Exp
            |  LT Exp Exp    |  PAR Exp
            |  ID String     |  NUM Int
            |  LET Decls Exp
data Decls  =  MANY Decls Dec
            |  ONE Dec
data Dec    =  DEC String Exp
```

```
data Type  =  INT | BOOL
```

How might we check an expression trees, as in figure 9, for well-formedness?





`type_check_exp[ADD(`t_1, t_2`)](st;type)`	`type_check_exp[VAR(str)](st;type)`
`:- type_check_exp[`t_1`](st;`$type_1$`)`	`:- retrieve(str;type).`
`type_check_exp[`t_2`](st;`$type_2$`)`	`check_add(`t_1, t_2`;INT)`
`check_add(`$type_1, type_2; type$`).`	`:-` t_1 `=INT,`t_2 `=INT.`

We need a symbol table as before ... but it is slightly more complex as it needs to hold type information for each declared variable:

$$[(x, int), (y, bool), ...]$$

As before, inserting adds the decalation to the front of list. A retrieval, given a string, will return a type or fail (recall the the example above retrieval merely succeeded or failed).

retrieve(x,[(x,int),(y,bool)];`INT`)	Succeeds and returns the type `INT`.
retrieve(z,[(x,int),(y,bool)];_)	Fails.

6.4 "Dance of the symbol table"

If the function types are also stored in the symbol table, what would the code look like if we wanted to build an intermediate representation. It actually means that we must thread the symbol table almost twice round the code: first to pick up the function names and types and second to handle the variable declarations.

The rest of these notes will sketch the salient aspects of defining an attribute system to calculate intermediate representation data for programs in a simple programming language. We'll begin

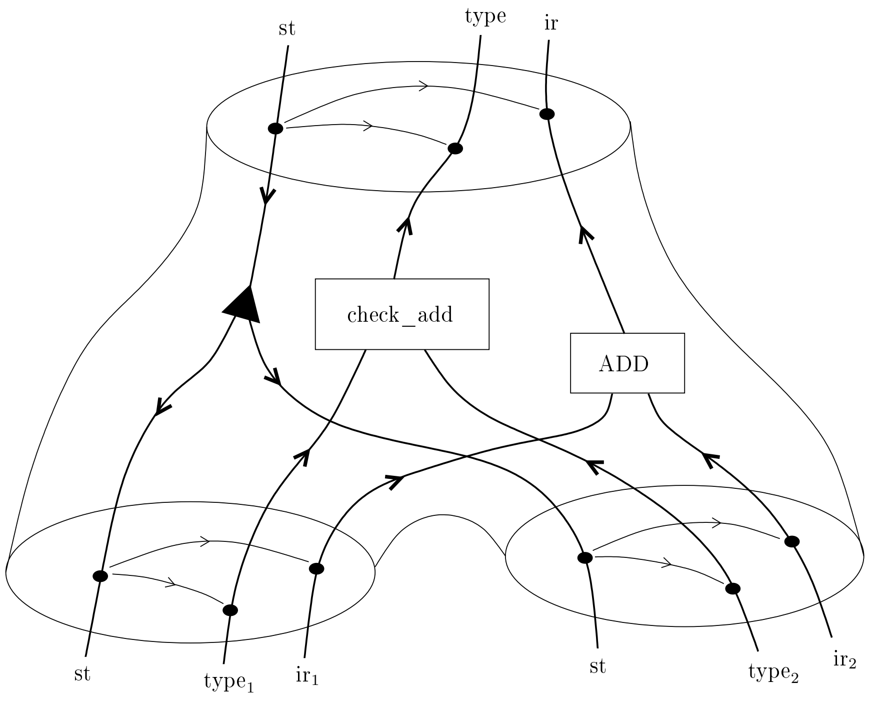


Figure 10: Returning a type and an intermediate representation for expressions

with a simple expression grammar, then introduce `let`-style local declarations, then types and type-checking, and finally scoped function declarations permitting in-block forward references. We drop the monadic Haskell exception model for the sake of more lucid code, with the understanding that exception handling can be restored essentially mechanically.

```
collects[decls](table;table′) :-
   genfuns[decls](table′;ir_fun),
   genstmts[stmts](table′;ir_stmts),
   ir := (ir_fun,ir_stmts).
```

```
ir decls stmts = (ir_fun,ir_stmts)
  where table’ = collects decls table
        ir_fun = genfuns decls table’
        ir_stmts = genstmts stmts table’
```

```
collects[decl::decls](table;table″) : −
  collect[decl](table;table′),
  collects[decl](table′;table″).
```

```
collects (decl:decls) table
  = collects decls $ collect decl table
```

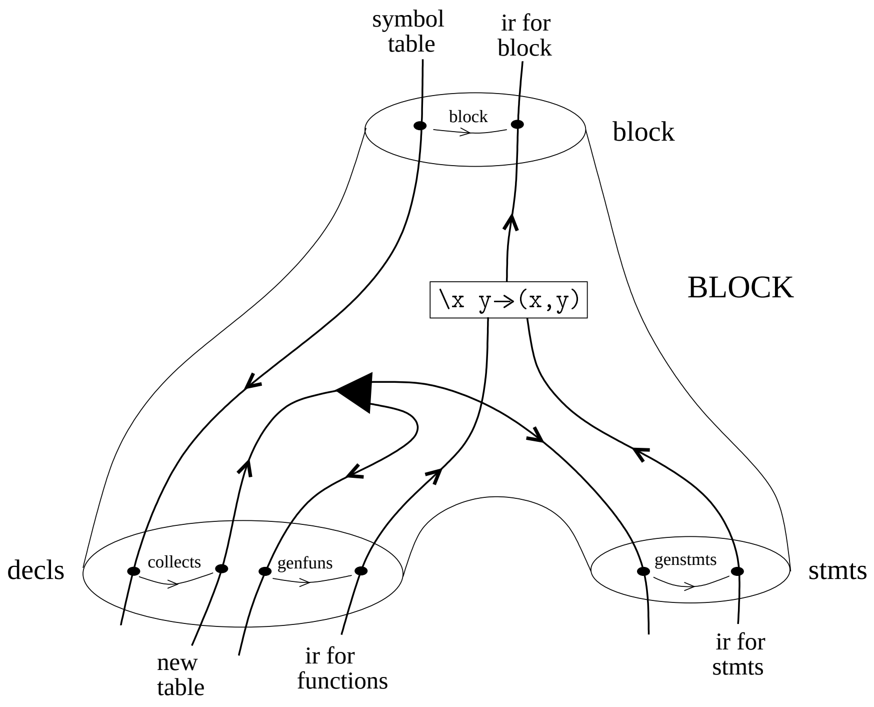


Figure 11: The symbol table dance for blocks

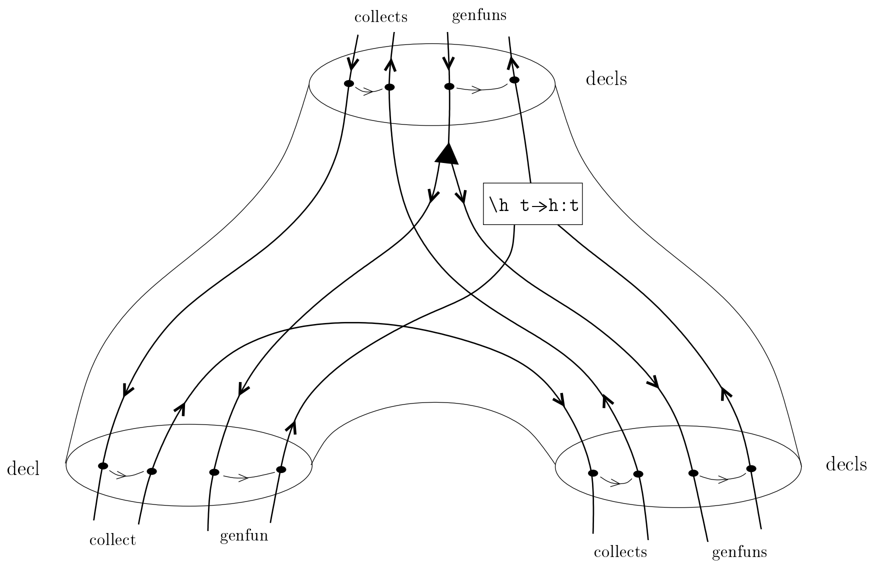


Figure 12: Tne symbol table dance for declarations

```
genfuns[decl::decls](table;ir_fun::ir_funs) : −
  genfun[decl](table;ir_fun),
  genfuns[decls](table;ir_funs).
genfuns[nil](table;nil).
```

```
genfuns [] table = []
genfuns (decl:decls) tab
  = (genfun decl tab,genfuns decls tab)
```

A Brief Guide to the Notation and Languages

These notes were originally written using *definite clause language* (DCL), which may add value for those who've seen some Prolog. We present Haskell translations of the DCG syntax in shaded boxes. Hence, most of the examples here are presented redundantly in three notational systems: The original DCG/Prolog, it's parallel Haskell translations, and the Plumbing Diagrams.

The following box summarizes the Haskell 'glue' needed to translate the spirit of the Prolog, the main feature of which is the exception monad `Okay` needed to model the exception raising semantics of DCL.

```
data Okay m = Okay m | NotOkay
testOkay ::  [Okay a] -> Okay a -> Okay a
testOkay xs y
  | allOkay xs = y
  | otherwise = NotOkay
allOkay lst = foldl isOkay True lst
isOkay (Okay _) = True
isOkay _ = False
fromOkay (Okay x) = x
fromOkay _ = error "fromOkay applied to NotOkay."
```

Haskell definitions we'll be using frequently.