CHAPTER 2

THE META-LANGUAGE

Chapters 2 and 3 are both concerned with the meta-language ("META-IV") in general; chapter 4 provides more motivation by showing how features of the meta-language are used in the denotational definition of programming languages. Readers who are new to semantic definitions are advised to read this chapter rather quickly and then use it as a reference after studying chapter 4. Those readers who are familiar with the Oxford work on denotational semantics will be struck by the large number of combinators used in "META-IV". The need for the extra "syntactic sugar" results from the definition of large systems.

The only knowledge assumed is of elementary set and logic notation -- thus, chapters 1-4, 14, and 15 of [Lipschutz 64a] would serve as adequate preparation. Lambda notation is introduced as a way of defining functions in general and combinators in particular. The notation for describing and manipulating objects is also described. The mathematical foundations of these concepts are reviewed in the next chapter.

A programmer's view of the meta-language is given in [Bjørner 78c]. (This chapter is a replacement for [Jones 78a].) Other related work includes that of the "Z" group [Abrial 80*], CLEAR [Burstall 77a, Burstall 80a], and several projects at SRI (Stanford Research Institute, Menlo Park, Ca., USA).

CONTENTS

2.1 DENOTATIONAL SEMANTICS

Chapter 1 introduces the fundamental ideas of denotational semantics: in order to define the semantics of some set of objects (L), a meaning function (M) is written which maps elements of L to some understood set of denotations (DEN); the structure of the elements of L is exploited in the rule that M should derive the denotation of structured elements of L (only) from the denotations of their components. Thus the meaning function is of type:

$M: L \rightarrow DEN$

and is defined by cases over the structure of L.

An example can be made of the definition of binary numerals (some notation is used in the examples which is defined only later in the chapter - it should, however, be clear enough to communicate the general idea). The binary digits are simply symbols and this fact can be emphasized by using rather clumsy names:

Bindigit = *0SYM* | *1SYM*

A binary numeral can either be a composite object or simply a digit:

Binnumeral = *Bincomposite* | *Bindigit*

A composite object is made up of two parts the first of which is a numeral and the second a digit:

Bincomposite :: *Binnumeral Bindigit*

The natural numbers can be used as denotations:

Nat0 = {0,1,2,...}

Thus the meaning of binary numerals is to be defined by a function of type:

M: Binnumeral → *Nat0*

This function can be defined by cases. For composite objects:

$M[mk\text{-}Bincomposite(n,d)] \triangleq M[n]*2 + M[d]$

Notice that this function builds the denotation (i.e. value) of the composite from the denotations of its components as required by the denotational method. The denotations of digits are given as follows:

$M[\underline{0SYM}] = 0$
$M[\underline{1SYM}] = 1$

Most definitions require a more complicated set of "understood Denotations". In particular, systems or languages normally require denotations which are functions over states. Thus, for a simple language:

$M: Lang \to TR$
$TR = STATE \tilde{\to} STATE$

(The distinction between total and partial functions is marked by superimposing a tilde on the arrow of the latter). It is, then, necessary to use notation for the creation and manipulation of functions: this is the topic of the next section.

2.2 FUNCTIONS

An example of the familiar style of function definition is:

$f(x) \triangleq x * x + 2$

When the argument and result sets are not obvious from context, a type clause can be written:

$f: Nat0 \to Nat0$

The use of conditional expressions in function definitions is familiar from many programming languages. For example, a function which yields the maximum of two integers is:

$max: Nat0 \times Nat0 \to Nat0$
$max(x,y) \triangleq \underline{if}\ x \leq y\ \underline{then}\ y\ \underline{else}\ x$

The "application" of a function $(D \to R)$ to an element of its argument set

(D) yields an element of the result set *(R)*, thus:

$f(3) = 11$
$max(2,4) = 4$

For various reasons, functions will frequently need to be recursive. A simple example of a function definition for square shows that the familiar style of presenting function definitions can be used:

$square: Nat0 \rightarrow Nat0$
$square(n) \triangleq \underline{if}\ n=0\ \underline{then}\ 0\ \underline{else}\ square(n-1) + 2*n - 1$

(Although the use of *square* within its own definition is familiar to programmers, it is necessary to consider below exactly what such a "definition" means).

But, when writing a denotational semantics definition, there will also be two further needs. Because of the need to create and manipulate functions as objects, a notation for unnamed functions is required. The "Lambda calculus" [Church 41a] is such a notation. The function named *f* above can be defined:

$\lambda x.x*x + 2$

This is an expression for the function. If it is required to name the function, it is possible to write:

$f = \lambda x.\ x*x + 2$

But the expression can be used without providing a name. For example, just as *f* can be applied to arguments, the unnamed function can be applied to values:

$(\lambda x.x*x+2)(3) = 11$
$(\lambda x.x*x+2)(1+3) = 18$

The form of such a "lambda expression" is:

$\lambda x.E$

in which *x* should be a name and *E* an expression (normally involving *x*).

The whole expression denotes the function which maps any argument to the value obtained by evaluating E when x equals the argument value.

Lambda expressions can be written which fulfil the need to create functions. Thus:

$$\lambda y.(\lambda x.x*x+y)$$

denotes a function which, for example, can be used to create the function considered above.

$$(\lambda y.(\lambda x.x*x+y))(2) = \lambda x.x*x+2$$

Functions of more than one argument (e.g. *max* above) can be reduced, by a process known as "Currying", to the simpler form:

$$\lambda x,y.E(x,y) = \lambda x.\lambda y.E(x,y)$$

Thus *max* can be defined:

$$max = \lambda x,y.\ \underline{if}\ x \leq y\ \underline{then}\ y\ \underline{else}\ x$$

In addition to defining functions whose values may be functions, lambda expressions can be written which take functions as arguments. For example:

$$\begin{aligned} &twice && = \lambda f.\lambda x.f(f(x)) \\ &twice(square) && = \lambda x.square(square(x)) \\ &(twice(square))(1) && = 1 \\ &(twice(square))(2) && = 16 \end{aligned}$$

The ability to define functions which take functions as arguments (known as "functionals") opens up a new way of understanding recursive "definitions". The definition of *square* given above can be viewed as an equation with *square* as an unknown (just as the quadratic:

$$2*x**2 = 14 - 3*x$$

is an equation with x as an unknown). If a new function is defined:

$$H = \lambda f.\lambda n.\underline{if}\ n=0\ \underline{then}\ 0\ \underline{else}\ f(n-1) + 2*n - 1$$

then *square* must be a solution to the equation:

$$square = H(square)$$

The fact that such solutions exist (and the choice of the "least fixed point" solution) is discussed in chapter 3. Also in need of mathematical foundations are the types of functionals. Chapter 3 explains why the familiar view of the set of all functions must be restricted when considering functions which take themselves as arguments.

In this chapter, the lambda notation is used on the assumption that it is sound (i.e. that the foundations are given). This section closes with some examples of the use of the rather general functions in the definition of programming languages.

(The first of these examples illustrates a use of recursion where proper "domains" -- see chapter 3 -- are required as denotations. In the *Binnumeral* example above, recursion was controlling a macro-expansion-like definition and the denotations could be ordinary sets.)

If a "while" statement is built according to the following syntax:

While :: *Expression Statement*
Statement = *Assign* |...| *While*

then a meaning function might be defined of type:

$$M: Statement \to STATE \rightharpoonup STATE$$

For this simple example it is assumed that expression evaluation causes no side-effects and that:

$$MX: Expression \to STATE \rightharpoonup Bool$$

where the defined elements of *Bool* are:

$$\{\underline{true}, \underline{false}\}$$

A recursive function can then be used to define the while case for M:

$$M[mk\text{-}While(b,s)](state) \triangleq$$
$$\quad \underline{let}\ wh = \lambda\sigma.(\underline{let}\ bv = MX[b](\sigma)\ \underline{in}$$
$$\quad\quad \underline{if}\ bv\ \underline{then}\ wh(M[s](\sigma))\ \underline{else}\ \sigma)\ \underline{in}$$
$$\quad wh(state)$$

The inner *let* introduces an abbreviation which is used *in* the following expression. The outer *let* introduces a name for a value which is defined recursively: some authors emphasize this by writing *letrec*. The possibility that the loop fails to terminate for some states is reflected by showing (with a tilde: ~) that the function is partial.

An example of the need for functions which take functions as arguments is found in languages which permit procedures to take procedures as arguments. The denotation of a procedure might be:

$$Procden = Argument^* \to STATE \xrightarrow{\sim} STATE$$

Arguments corresponding to "by value" arithmetic parameters might be numbers; those corresponding to "by reference" parameters might be locations; but the arguments corresponding to procedure parameters must be elements of *Procden*. Both of the issues raised here are considered further in chapter 3.

2.3 COMBINATORS

It would be possible to write out complete language definitions using the lambda notation. Meaning would be defined by a translation into a large lambda expression. For small languages, this is sometimes done. For larger languages, a much more readable definition can be given by using some "combinators" which correspond to commonly used ways of combining function values. For example, given functions:

$$f: D1 \to D2$$
$$g: D2 \to D3$$

then their "composition" is:

$$g \circ f = \lambda x.g(f(x))$$

Notice that f is applied first. Thus:

$g \circ f: \quad D1 \rightarrow D3$

Since this is defined in terms of the Lambda calculus, no new foundation problems are introduced: the combinators simply provide a more perspicuous notation for writing a lambda expression.

One of the differences between definitions written by the "Oxford" and "Vienna schools", in the greater use of combinators by the latter. This difference has, at least in part, resulted from the fact that the Vienna group faced the task of defining rather large languages and systems. Just as in programming, the larger the final text the greater the justification for defining concepts of general use. Because the combinators can be (and are in this section) defined in terms of Lambda calculus, this difference between the two schools can be seen to be superficial.

The order of composition is the reverse of that which is frequently natural in a semantic definition. It is therefore convenient to define a combinator in which the first operand is applied before the second. This "semicolon" combinator is such that:

$f: D1 \rightarrow D2$
$g: D2 \rightarrow D3$
$f;g = g \circ f$

Thus:

$f;g: D1 \rightarrow D3$

In particular, where both operands are of type $STATE \rightarrow STATE$, so is their combination. This can be used, for example, within the definition of the meaning of a while construct given above to write:

$\underline{if}\ bv\ \underline{then}\ (M[s];wh)\ \underline{else}\ I_{STATE}$

where I_{STATE} is the identity function on states.

A small extension to the same example shows the desirability of another combinator. Suppose expression evaluation is allowed to create side-effects, then the type of the expression meaning function would be:

$MX: Expression \rightarrow STATE \xrightarrow{\sim} STATE \times VAL$

The use of the semicolon above has removed the need to write the σ for part of the definition: how can this be extended to the whole definition with the above type for *MX*? A "define" combinator can be given:

$$f : D1 \to D2 \times D3$$
$$e(x) : D2 \to D4, \quad x \in D3$$
$$(\underline{def}\ x: f; e(x)) : D1 \to D4$$
$$(\underline{def}\ x: f; e(x)) = f; (\lambda\sigma, x.e(x)(\sigma))$$

In particular:

$$(\underline{def} x: \_; \_): (STATE \xrightarrow{\sim} STATE \times VAL) \times (VAL \to STATE \xrightarrow{\sim} STATE) \to (STATE \xrightarrow{\sim} STATE)$$

The association rules for → and × are that × binds more strongly than → and is associative, and that → is right associative. Thus:

$A \times B \to C \to D \times E$ is the same as: $(A \times B) \to (C \to (D \times E))$

The complete definition of the meaning of the while construct can now be given:

$$M[mk\text{-}While(b,s)] \triangleq$$
$$\underline{let}\ wh = (\underline{def}\ bv: MX[b];$$
$$\underline{if}\ bv\ \underline{then}\ M[s]; wh\ \underline{else}\ I_{STATE})\ \underline{in}\ wh$$

The elimination of parameters (i.e. *state*, σ) corresponding to states in this example has made the definition clearer. The difference is even more significant on larger examples.

Some cases of such a transformation may determine a value without changing the state. The *return* combinator can be used to promote a simple value to a transformation. Thus for $v \in VAL$:

$$\underline{return}\ v: STATE \xrightarrow{\sim} STATE \times VAL$$

The conditional used in the definition of the while statement is also a simple combinator. It should be noticed that this use is "dynamic" in the sense that the result is determined by a value which is created by one of the transformations. There are other uses of conditionals which depend only on the "static" text whose denotation is being defined. Another useful static combinator is *for*. Thus in defining the meaning of

a list of statements in a *compound* statement:

```
Statement = Assign | Compound | ... | While
Compound  :: Statement*
```

it is possible to write:

```
M[mk-Compound(sl)] Δ for i=1 to lensl do M[sl[i]]
```

Rather than using recursion, as in:

```
M[mk-Compound(sl)] Δ
    if lensl=0 then I_STATE
    else M[hdsl];M[mk-Compound(tlsl)]
```

A similar static expansion can be used for *def*:

```
def vl: <M[al[i]] | 1≤i≤lenal>; ... vl ...
```

is the same as:

```
def vl1: M[al[1]];
def vl2: M[al[2]];
:
:
def vln: M[al[lenal]];
let vl = <vl1,vl2,...,vln> in ... vl ...
```

A language or system without any exception type constructs could be defined using only the above combinators. The handling of exception constructs presents problems for the denotational method precisely because the effect of an exception cuts across the structure over which the denotations are supposed to be constructed. The archetypal construct in this area is the goto statement. The difficulty, is to provide a denotation for simple statements which can be used to derive the denotation for sequences of statements. The definition of goto statements provides, in chapter 4, the motivation for the *exit* approach used in VDM. Chapter 5 contrasts and connects this approach with the "continuations" used by the Oxford school. Here, the combinators relating to the *exit* approach are defined for reference. Readers should probably skip the remainder of this section at first reading.

The basic idea is to use denotations which are capable of reflecting the exception. The result of applying the denotation to a particular state is a state plus an exception indication: a *nil* value indicates the lack of an exception whereas information about the exception is given in non-*nil* values. Combinators are defined which simplify the move from transformations of type:

$$T = STATE \xrightarrow{\sim} STATE$$

to ones which can reflect any abnormal result:

$$TR = STATE \xrightarrow{\sim} STATE \times [ABNORMAL]$$

The set *ABNORMAL* is chosen to fit the system being defined; here only the distinction between *nil* and non *nil* is of importance. The basic *exit* combinator causes a non *nil* value to be returned with an unchanged state. Thus:

$$v \in ABNORMAL$$
$$exit\ v: TR$$
$$exit\ v = \lambda\sigma.(\sigma, v)$$

The handling of non *nil* abnormal values is defined by a *tixe* combinator. For:

$$m: ABNORMAL \xrightarrow{m} TR$$
$$f: TR$$

then:

$$(tixe\ m\ in\ f): TR$$
$$(tixe\ m\ in\ f) = (let\ r = (\lambda\sigma, a.if\ a \in dom m\ then\ r(m(a)(\sigma))\ else\ (\sigma, a))\ in\ r \circ f)$$

The semicolon combinator is redefined to reflect the need to propagate abnormal returned values. With:

$$f: TR$$
$$g: TR$$

then:

$f;g: TR$
$(f;g) = (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ g(\sigma)\ \underline{else}\ (\sigma,a))\circ f$

Similar changes are made to the *def* combinator. Where a simple transformation (T) occurs in a context of an exit transformation, it is automatically interpreted as returning a *nil* abnormal component. Thus:

$f: STATE \rightarrow STATE$

is interpreted, in appropriate contexts as:

$\lambda\sigma.(f(\sigma),\underline{nil})$

One other combinator relating to *exit* transformations takes:

$f: STATE \rightarrow STATE$
$g: TR$

to give:

$(\underline{always}\ f\ \underline{in}\ g) : TR$
$(\underline{always}\ f\ \underline{in}\ g) = (\lambda\sigma,a.(f(\sigma),a))\circ g$

The combinators given in this section are not a fixed set for the meta-language. Others may be defined in the same way if there is a need. This collection suffices however for the language definitions given in this book. Certain others are used in the "systems definitions" in part III - they are defined in the "Glossary of Notation".

2.4 LOGIC NOTATION

The symbols to be used for the propositional operators are $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), $=$ (equivalence). The operators have been given in decreasing order of priority.

The truth values are:

$Bool = \{\underline{true},\underline{false}\}$

The quantifier symbols are $\forall$ (universal) and $\exists$ (existential). Quantified

expressions will normally be bounded. For example:

$(\forall x \in Nat)(is\text{-}even(2*x))$
$(\exists x \in \{11,12,13\})(is\text{-}prime(x))$

The careful use of bounds can avoid using operators on values for which they are not defined. Thus we write:

$(\forall x \in Int - \{0\})(p(1/x))$

rather than:

$(\forall x)(x \neq 0 \supset p(1/x))$

Where bounded quantifiers are insufficient, conditional expression can be used. For example:

$\underline{if}\ x=0\ \underline{then}\ q(y)\ \underline{else}\ p(1/x)$

rather than:

$x=0 \wedge q(y) \vee x \neq 0 \wedge p(1/x)$

2.5 OBJECTS

Thus far in the discussion of semantics the structure of the states has not been considered. Given the careful foundations which are provided for general functions, it would be possible to view most semantic objects as functions. For example, store can be viewed as a function from locations to values - even a subset of some given set can be viewed as a function from the given set to the set of Boolean values. Such a view would, however, limit the operations which can be applied. The domain of a function is, for example, undecidable in general and it would not be obvious that it was sound to write an expression involving the domain of store. For this reason, authors of VDM specifications carefully distinguish "sets", "maps", and "lists" from more general functions. Objects to be used in definitions are restricted as follows: "sets" are finite and contain only distinguishable elements (not functions for example); "maps" are finite functions which are constructed in ways which ensure that their domains are apparent; "lists" are also finite.

The class of all finite subsets of some given set X is written:

$$X\text{-}set$$

Notice that if X is infinite, this is a proper subset of the power set since only finite sets belong to the class $X\text{-}set$.

The operators involving sets are shown in the "ADJ" diagram in figure 1. These diagrams (cf. [Goguen75a]) show the types in ovals and fix the type of each operator by the arcs. (The subset operator, e.g. is between two operands each of type set and yields a Boolean result.)

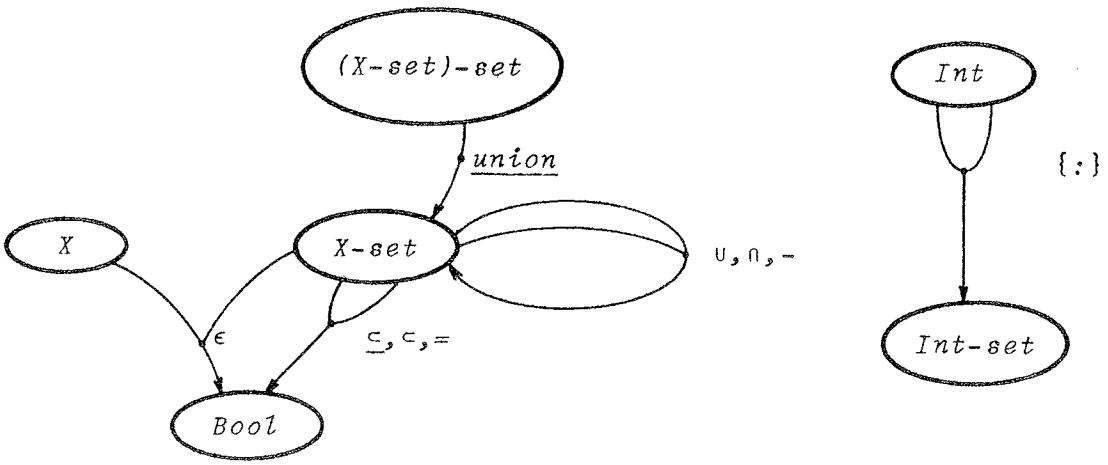

Fig. 1: Set operators

Notice that the distributed union operator is written:

$$\underline{union}\ SS = \{e \mid (\exists S \in SS)(e \in S)\}$$

The shorthand for a set of integers is:

$$\{i:j\} = \{k \in Int \mid i \leq k \leq j\}$$

The basic sets used in this book are:

$\{\}$ = empty set
$Bool = \{\underline{true}, \underline{false}\}$
$Nat = \{1, 2, \ldots\}$
$Nat0 = \{0, 1, 2, \ldots\}$
$Int = \{\ldots, -1, 0, 1, \ldots\}$

The class of functions which satisfy the constraints of maps are defined:

$$D \overset{m}{\to} R$$

with obvious extensions for domains which are Cartesian products. One-one maps are marked:

$$D \overset{1m}{\to} R$$

The operators involving maps are shown in figure 2.

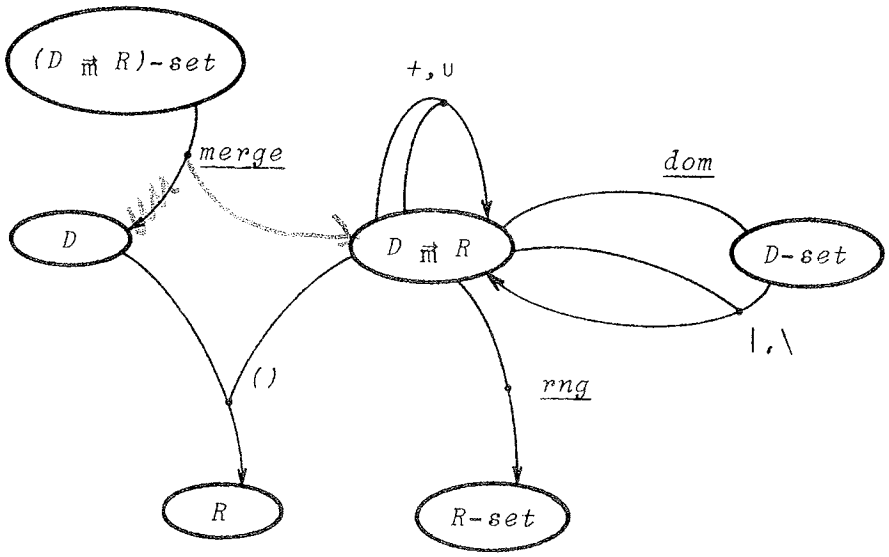

Fig. 2: Map operators

The empty map is written:

$$[\,]$$

Other maps can either be written explicitly:

$$[a \mapsto b,\ b \mapsto c,\ c \mapsto a]$$

or implicitly:

$$[x \mapsto x! \mid x \in \{1:4\}] = [1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 6,\ 4 \mapsto 24]$$

Application is written exactly as for functions.

Assuming domain and application (over the domain) are understood, the remaining operators can be defined. The range of a map is:

$$\underline{rng}M = \{M(d) \mid d \in \underline{dom}M\}$$

The map "overwrite" operator is defined:

$$M1 + M2 = [d \mapsto (\underline{if}\ d \in \underline{dom}M2\ \underline{then}\ M2(d)\ \underline{else}\ M1(d)) \mid d \in (\underline{dom}M1 \cup \underline{dom}M2)]$$

Map "union" is only defined if the domains of the maps are disjoint. Its definition is otherwise the same as for overwrite. The advantage of using a separate operator for this special case is that it is commutative. A map can be "restricted":

$$M \mid S = [d \mapsto M(d) \mid d \in (\underline{dom}M \cap S)]$$

or a set of domain elements can be removed:

$$M \setminus S = [d \mapsto M(d) \mid d \in (\underline{dom}M - S)]$$

The *merge* operator provides a distributed form of map union. Assuming:

$$(\forall m_1, m_2 \in ms)(m_1 = m_2 \vee \underline{dom}m_1 \cap \underline{dom}m_2 = \{\})$$

then $\underline{merge}ms$ gives the map whose domain is the union of the domains of the individual maps and whose results match that from the appropriate individual map.

Lists (or Tuples) could be viewed as maps from a set of natural numbers. Reasoning about lists can, however, be aided by providing operators which are intuitively obvious. The list operators are shown in figure 3, next page. The class of all finite lists is defined by X^* and all non-empty lists by X^+.

Here again taking two operators (*len* and application) as basic makes it possible to define the others. The set of indices of a list is:

$$\underline{inds}L = \{1 : \underline{len}L\}$$

The set of elements in a list is:

$$\underline{elems}L = \{L[i] \mid i \in \underline{inds}L\}$$

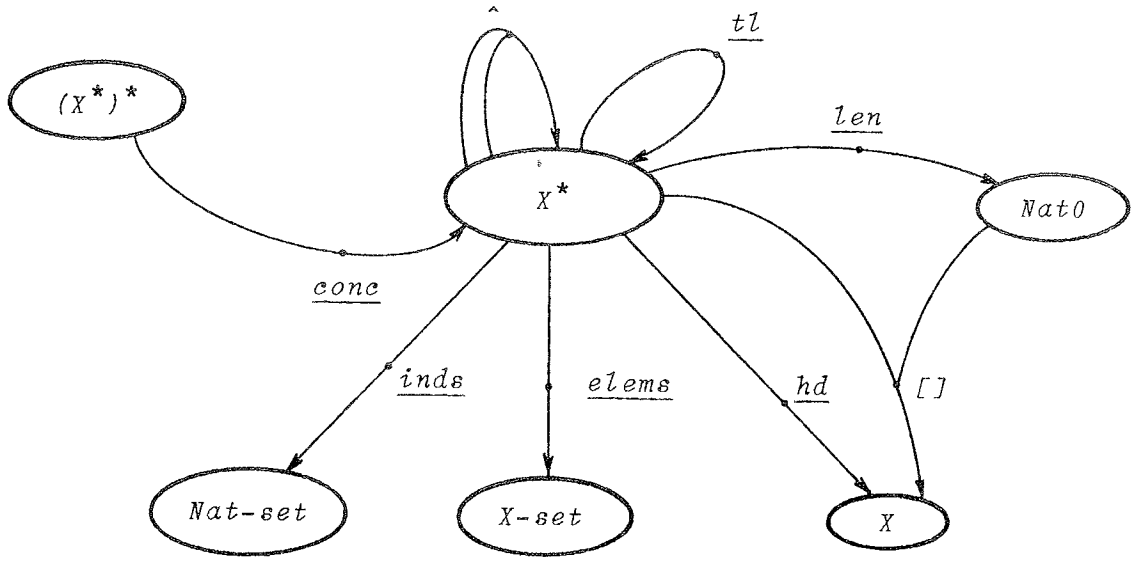

Fig. 3: List operators

The first element in a list is given by:

$$\underline{hd}L = L[1]$$

The list remaining when the head has been removed is given by:

$$L' = \underline{tl}L \supset \underline{len}L' = \underline{len}L-1 \wedge (\forall i \in \underline{inds}L')(L'[i] = L[i+1])$$

Notice that neither the head nor the tail operators are defined on the empty list. Concatenation of two lists is defined:

$$L = L1\hat{}L2 \supset$$
$$\underline{len}L = \underline{len}L1 + \underline{len}L2 \wedge$$
$$(\forall i \in \underline{inds}L1)(L[i] = L1[i]) \wedge$$
$$(\forall i \in \underline{inds}L2)(L[i+\underline{len}L1] = L2[i])$$

Distributed concatenation is defined:

$$\underline{conc}LL = \underline{if}\ LL=\langle\rangle\ \underline{then}\ \langle\rangle\ \underline{else}\ \underline{hd}LL\ \hat{}\ \underline{conc}(\underline{tl}LL)$$

2.6 DEFINING OBJECTS

In writing a specification, both elementary and structured objects are required. Most of the ways of structuring objects (classes of maps, etc.) are given above. Thus if:

$$M = Int \overset{m}{\to} Bool$$

then:

$$\{[\,],[\,1 \mapsto \underline{true}\,],[\,2 \mapsto \underline{false},\ 7 \mapsto \underline{true}\,]\} \subset M$$

An "abstract syntax" notation is given below. "Elementary objects" are those whose structure is of no interest for a particular specification. These can be elements of *Token* or *Quot*. *Token* is an infinite set of objects whose representation is not exposed: the identifiers in a programming language would normally be treated as tokens. Where there is a need to enumerate specific elementary objects, quotations are written as sequences of underlined (usually upper-case) letters (e.g. A, ABC, LABEL).

The most extensive object definitions to be written are for "abstract syntax". It is therefore suggestive to make the rules look similar to concrete syntax (e.g. BNF notation). Thus:

$$Stmt = Assn \mid Goto$$

(Notice that this is a simple, nondiscriminated, union.) Similarly, the convention of using square brackets for an optional element is adopted. Thus:

$$[X] = X \cup \{\underline{nil}\}$$

The *nil* object showing absence of the optional X.

Because of the use in abstract syntax, there is a need to build "record" type composite objects which have inhomogenous fields. For example:

$$Assign :: Varref\ \ Expr$$

defines that any element of the set of objects named *Assign* has two components, the first of which is an element of the set named *Varref* and the second an element of the set named *Expr*. The set of elements of a class defined by a "constructor" rule is, for the above example:

$$Assign = \{mk\text{-}Assign(vr,e) \mid vr \in Varref \wedge e \in Expr\}$$

The constructor function:

mk-Assign: Varref × *Expr* → *Assign*

can be thought of as installing a hidden flag which ensures that sets defined by different (constructor) rules are disjoint. Formally, it is sufficient to know this uniqueness property without showing how the elements are flagged. (The use of such constructor rules obviates the need to have a disjoint union operator.)

Given constructed objects, it is frequently necessary to decompose them. One way of doing this is by writing the constructor in a "left-hand-side" position (i.e. on the left of a definition or as a parameter). Thus:

let mk-Assign(vr,e) = s in ... e ... vr ...

defines *vr* and *e* to have the values of the appropriate components of the (previously defined) value *s*.

As an additional convenience the components of a constructed object can be named. For example:

Assign :: s-lhs:Varref s-rhs:Expr

These "selector" names can be used as functions to obtain the components of a constructed object. Thus:

s-lhs: Assign → Varref
s-rhs: Assign → Expr

s-lhs(mk-Assign(vr,e)) = vr
s-rhs(mk-Assign(vr,e)) = e

The constructor rules and definitions which use *X-set* etc. can be used recursively: in any such use the valid objects are all finite instances satisfying the (syntactic) equations.

Where a state is itself a constructed object, it is permissible to use the selector names like variables in an assignment combinator. Thus:

*(*Fl *:= e(c* Fl*)): STATE → STATE*
*(*Fl *:= e(c* Fl*)) = λσ.(mk-STATE(e(F1(σ)),F2(σ),...))*

Notice that the use of the reference on the right hand side of the assignment is marked with a contents *(c)* operator. This is more explicit than in programming languages where the use of a name has a different meaning depending on context.

2.7 ON NON-DETERMINISM

Non-determinism is something which need be considered in many specifications. For example, in most programming languages (including ALGOL 60) the order of access to variables within expressions is not defined. Thus in:

$$a + (b + c)$$

the variables can be accessed in any one of six orders. Notice that order of access is separate issue from the order in which the operators are applied. The problem is that if some sub-expressions include references to functions which cause side-effects, the value of the variables might differ depending on the order of access. Why should the designers of a language leave such an odd non-determinism unresolved? It is reasonable to assume that this was done to permit optimization like common sub-expression elimination. There is a warning that users of ALGOL should not write programs which depend on a particular order. But then there are also many languages which contain features which deliberately introduce nondeterminism (cf. "guarded commands" in [Dijkstra 75a]). One approach to the semantics of non-determinism is to use relations on states as the denotations (cf. [Park 80a], [Jones 81a]). This approach cannot, unfortunately, be extended to cover procedures which can take themselves as arguments. An alternative approach is to use "power domains" (cf. [Plotkin 76a], [Smyth 76a]). The topic of specifying non-deterministic systems is not covered in this book.

There is, however, some non-determinism used in the specifications given below. In certain places it can aid the proof of correctness of an implementation to show that the result is not affected by a particular choice. For example, the choice of a free store location should not be fixed by the specification. Even to hypothesize a choice function requires fixing what influences the choice. Showing the possible results makes it easier to show that an implementation is correct. Strictly, however, each such non-deterministic choice should be accompanied by a

proof that the non-determinism has no effect on the overall outcome. This is necessary to justify the use of functional (rather than relational or power) domains.