

The Reference Languages

=====

We will describe two languages, the complete reference language and the proper reference language. The proper reference language is supposed to be easy to explain and the complete reference language is supposed to be very readable. We want to the relationship between the proper and complete reference language to satisfy two principals:

- 1) The proper language is a "sublanguage" of the complete language.
- 2) Every construct of the proper language means the same when viewed as a construct in the complete language.
- 3) Every construct in the complete language translates EASILY into a unique construct in the proper language.

The Whole Show

=====

The language consists of a single expression. There are auxiliary things like modules to aid in the writing down of that single expression.

Atoms

=====

We want a large, rich set of atoms. In order to achieve this, we will allow any arbitrary sequence of keystrokes to be an atom. We informally define a keystroke to be any roman letter or special symbol that can be typed. Of course, every atom is an object.

When writing down an atom, we need a way of distinguishing atoms from punction marks. So we define an atom in the proper language to be anything of the form !...! where ... represents any sequence of keystrokes. Since we want to be able to have the ! to appear in atoms, we need an escape mechanism for distinguishing the use of ! as a keystroke in an atom from the use of ! as the end bottomimiter of the atom. We use the keystroke \ for this purpose. To improve readability in the complete language, we will allow any atom that contains only letters and digits to be written using italic fonts.

Examples:

=====

```
!This is an atom!
!123abc!
! !
! !
!!
!\!!
```

Characters

=====

All characters are atoms. A character is merely an atom that can be typed with a single keystroke, not counting the escape character \.

There is a primitive function Char that is the characteristic predicate for characters.

Examples:

=====

```
!A!
!\!!
! !
```

Strings

=====

A string is merely a sequence of characters. In the proper language it must be written as such. In the complete language, we allow the shorthand '...' where ... is any sequence of characters. Once again, we use \ as an escape mechanism, but this time it is an escape mechanism for the use of '.

Complete Language Example 90-WAR-998 Proper Language Counterpart

```

< !a! , ! b! >
'ab'
'\b'
'!c'
''

```

```

< !a! , !b! >
< !a! , !b! >
< !?!, !b! >
< !\! , !c! >
< >

```

There are two primitive functions StringToAtom and AtomToString. StringToAtom converts strings into atoms (producing bottom if not given a string) and AtomToString converts atoms into strings. Note that the function String = and o alpha Char.

Functions

=====

Just as with atoms, we want to have a rich set of function names. We allow a function name to be an arbitrary sequence of keystrokes, bottomimited by ". In the complete language, we allow function names to be typed in Roman letters if there are no special characters in the name. There are of course, no functions like StringToAtom and AtomToString for function names.

Complete Language Example
=====

```

abc
"X*y"

```

Proper Language Counterpart
=====

```

"abc"
"X*y"

```

Patterns

=====

A pattern has one of the forms:

fn n1.p2
fn n1.
e1
e1 p1 , ... , pn]
p1 .@ p2
p1 @. p2
p1 and p2?

name of func
fn name name(P1)

name(P1) U ... U name(Pn)
N(P1) U N(P2)

where n1 is a function name, e1 is an expression (to be defined shortly), and p1, p2, ... pn are patterns that have no function names (immediately to the left of a dot) in common. *such that N(P1) ... N(Pn) on null display*

(A pattern has two parts, a function part and a definition part. The function part acts just like an ordinary function. The definition part acts like a list of definitions. We give the cases in the chart below. We use the convention that fi is function part of pi and that Di is a typical element of the definition list of pi. di is the new definition associated with the name given in Di. Also note that g1 land g2 is the function "length >= 1" and g1 and g2.

Pattern =====	Function Part =====	Definition Part =====
n1.p2	f2	def n1 = id , d2 = D2
n1.	~1	def n1 = id
e1	e1	No definitions
e1 p1 , ... , pn]	e1 f1 , ... , fn]	def ... di = Di o i ...
p1 .@ p2	f1 o 1 land f2 o t1	def d1 = D1 o 1 , d2 = D2 o t1
p1 @. p2	f1 o t1r land f2 o -1	def d1 = D1 o t1r , d2 = D2 o -1
p1 and p2	f1 and f2	def d1 = D1 , d2 = D2

DefList

=====

A deflist is just a list of definitions of the form:

```

def f1 = e1
def f2 = e2 ...

```

where e1, ... are arbitrary expressions and f1, f2, ... are all distinct function names.

Expressions

=====

All expressions in the proper language have one of the forms:

FunctionName

```

e1 : e2
< e1 , ... , en >
( p1 => e1 ; e2 )

```

where e1, e2, ... , en are expressions, p1 is a pattern.

Atoms, of course, mean themselves. The meaning of the function names for the primitive functions are given in the section on primitive functions. The meaning of all other function names are given either by patterns or the module (cf. the definition section). The meaning of e1 : e2 is just the applications of the meanings. The meaning of a sequence is the sequence of the meanings.

The meaning of the conditional is a function such that when applied to x yields a value by first computing the function part of p1 applied to x and then based on that value returning bottom if p1 : x = bottom, e2 : x if p1 : x = del, and a value based on e1 and x otherwise.

Assume the pattern p1 defines the names v1 , ... , vn to be f1 , ... , fn respectively. Also assume that e1 is E (v1 , ... , vn). the value of the => conditional is E (f1 : x , ... , fn : x).

This completes the description of the proper reference language.

Abbreviations

=====

We present some constructs in the complete language and their abbreviations in the proper language. One construct that has been added to the complete language is an infix conditional with definitions. It has the form (p1 also deflist -> e2 ; e3) where e2 and e3 are expressions and p1 is a pattern. It has the proper language form given below. It also makes some new definitions. The pattern p1 makes some definitions (cf. patterns) and the deflist makes some new definitions. The scope of both the pattern and the deflist definitions is the deflist and e2 (NOT e3). As usual, it is illegal for a function name to be defined twice (in the union of the pattern and deflist definitions).

Complete language expression

=====

```

f o g
[ f1 , ... , fn ]
( p1 also deflist -> e2 ; e3 )

```

```

e[ f1 , ... , fn ]

```

Proper language representation

=====

```

comp : < f , g >
cons : < f1 , ... , fn >
cond : < f1 , e2 , e3 >
      where f1 is the function part
            of the pattern p1.
see section on Filters.

```

965 - IBM - 06

We list the primitive functions and how they act.

In the following, we assume that no x_i or y_i is bottom.

All our functions have an implicit first clause

$f : (\text{bottom} , h) = (\text{bottom} , h)$ so that all our functions are strict.

Furthermore, if the argument coming in fails to meet the form specified, then the result is (bottom , h) where h was the incoming history.

$al : (\langle y , \langle x_1 , \dots , x_n \rangle \rangle , h) = (\langle y , x_1 , \dots , x_n \rangle , h)$

$ar : (\langle \langle x_1 , \dots , x_n \rangle , y \rangle , h) = (\langle x_1 , \dots , x_n , y \rangle , h)$

$i : (\langle x_1 , \dots , x_n \rangle , h) = (x_i , h)$ if $0 < i \leq n$

$i : (\langle x_1 , \dots , x_n \rangle , h) = (x_j , h)$ if $i < 0 < j = n + 1 + i$

$tl : (\langle x_1 , \dots , x_n \rangle , h) = (\langle x_2 , \dots , x_n \rangle , h)$ if $1 < n$

$tlr : (\langle x_1 , \dots , x_n \rangle , h) = (\langle x_1 , \dots , x_j \rangle , h)$ if $0 \leq j = n - 1$

$app : (\langle x_1 , x_2 \rangle , h) = x_1 : (x_2 , h)$

$K : (x_1 , h_1) = (g , h_1)$ where $g : (x_2 , h_2) = (x_1 , h_2)$

$id : (x_1 , h) = (x_1 , h)$

$comp : (\langle x_1 , x_2 \rangle , h_1) = (g , h_1)$
 where $g : (x_3 , h_3) = x_1 : (x_2 : (x_3 , h_3))$

$cons : (\langle x_1 , \dots , x_n \rangle , h) = (g , h)$ where given y, h_0
 we define $(z_j , h_j) = x_j : (y , h_{j-1})$ for $0 < j \leq n$ and where
 $g : (y , h_0) =$
 (bottom , h_1) if $z_i = \text{bottom}$ and $z_j \text{ NE bottom}$ for $j < i$
 $(\langle z_1 , \dots , z_n \rangle , h_n)$ if $z_j \text{ NE bottom}$ for $j \leq n$.

$cond : (\langle x_1 , x_2 , x_3 \rangle , h) = (g , h)$ where
 $g : (y , h_0) =$
 (bottom , h_1) if $x_1 : (y , h_0) = (\text{bottom} , h_1)$
 $x_3 : (y , h_1)$ if $x_1 : (y , h_0) = (\text{omega} , h_1)$
 $x_2 : (y , h_1)$ if $x_1 : (y , h_0) = (\text{otherwise} , h_1)$

$null : (\langle \rangle , h) = (\text{omega} , h)$ and

$null : (\langle x_1 , \dots , x_n \rangle , h) = (\langle x_1 , \dots , x_n \rangle , h)$ if $0 < n$

$and : (x , h) = (\text{bottom} , h)$ if x is not a sequence

$and : (\langle x_1 , \dots , x_n \rangle , h) = (\text{del} , h)$ if any of the $x_i = \text{del}$

$and : (\langle x_1 , \dots , x_n \rangle , h) = (\langle x_1 , \dots , x_n \rangle , h)$ otherwise

$or : (x , h) = (\text{bottom} , h)$ if x is not a sequence

$or : (\langle x_1 , \dots , x_n \rangle , h) = (\text{omega} , h)$ if all of the $x_i = \text{del}$

$or : (\langle x_1 , \dots , x_n \rangle , h) = (\langle x_1 , \dots , x_n \rangle , h)$ otherwise

FILTERS.MAN

Predicates

=====

Formally, every function is a predicate. When a speaker uses the word "predicate" he is displaying his state of mind rather than something inherent about functions. In particular, when he is thinking of f as a predicate, he is thinking of $f:x = \text{bottom}$ means the answer is bottom, $f:x = \text{del}$ means the answer is "false", and $f:x = \text{anything else}$ means the answer is "true".

Sieves

=====

A sieve is a function such that $f : x$ is an element of $\{ x , \text{del} , \text{bottom} \}$ for all values of x and such that $f : \text{del} = \text{del}$.

Filters

=====

A filter is a function such that $f : x$ is an element of $\{ x , \text{bottom} \}$ for all values of x .

Delta

=====

```
def1 crash = ( id -> id ; ~bottom ).
def2 delta = ( f. => ( f -> id ; ~bottom ) ; ~bottom ).
```

E-Construction

=====

For each integer n , e -construction is in the complete language and is an abbreviation for the proper language expression that abbreviates the expression below.

```
e[ f1 , ... , fn ] =
  ( Seq ->
    ( length = ~n -> and o [ f1 o 1 , ... , fn o n ] ; ~del ) ;
    ~del ).
```

965 - IBM - 06

We briefly explain how to do datatypes.

OUTSIDE VISIBLE section.

In this section, the data typer writes the signatures and axioms that will be visible to the outside. The user of data types will be able to use this information. Conceptually, the user will know that semantically an element of data type X will be $X\langle \text{something} \rangle$ but he will NOT know what that something is. Furthermore, he will not be able to do any operations on anything of the form $X\langle \dots \rangle$ except as provided by the data typer.

INTERNAL DECLARATIONS section.

The data typer will write a declaration

LIKE stack is seq.

The system will automatically create two functions StackToSeq and SeqToStack that will be bijective functions from sequences to sequences labelled by the atom stack. StackToSeq o SeqToStack will be the identity on Sequences and SeqToStack o StackToSeq will be the identity on Stacks.

EXTENSIONS.

Here are some issues.

The data typer may wish to extend functions like plus, times, eq. Should we allow this extension? (If we do, I propose that the data typer say EXT instead of def). The problem with allowing these extensions is that they might destroy certain algebraic laws. For example, plus is commutative yet the extension given by the data typer might not be commutative. Also, the extension to eq might not be transitive, etc.

A note about eq.

eq is treated like any other primitive function (eg, al). In particular, the relationship between the function eq and the fundamental = on the underlying semantic domain is a relatively weak one. In particular, the following seem to be the case:

- 1) If $x = y$, then $\text{eq} : \langle x, y \rangle$ is either $\langle x, y \rangle$ or bottom.
(This can be destroyed by a bug in the data type causing eq not to be reflexive.)
- 2) The language is referentially transparent with respect to =.
Ie, if $x = y$ then $E(x) = E(y)$.
(This can NOT be destroyed by the data type.)
- 3) The language is WEAKLY referentially transparent with respect to eq.
Ie, if $\text{eq} : \langle x, y \rangle = \langle x, y \rangle$, then
 $\text{eq} : \langle E(x), E(y) \rangle = \langle E(x), E(y) \rangle$ or bottom.
(This can be destroyed by a bug in the data type.
For example, this will be destroyed if the data typer implements sets as sequences with duplicates allowed, and the cardinality function just counts elements of the sequence. In this case,
 $\text{eq} : \langle \text{set}\langle 1,1 \rangle, \text{set}\langle 1 \rangle \rangle = \langle \text{set}\langle 1,1 \rangle, \text{set}\langle 1 \rangle \rangle$ but
 $\text{card} : \text{set}\langle 1,1 \rangle = 2$ and $\text{card} : \text{set}\langle 1 \rangle = 1$ so that
 $\text{eq} : \langle \text{card} : \text{set}\langle 1,1 \rangle, \text{card} : \text{set}\langle 1 \rangle \rangle = \text{del}.$)
(Note that referential transparency does NOT hold for the language even without data types. For example,
 $\text{eq} : \langle 3, 3 \rangle = \langle 3, 3 \rangle$ but
 $\text{eq} : \langle K : 3, K : 3 \rangle = \text{bottom}.$)

In this section, we outline the operational semantics of the complete language.

Step 1: Make sure that all function names have at most one definition.

- 1a) For each use of a function name, identify the corresponding definition. Primitive functions names are NOT allowed to be redefined so the use of a primitive function name always refers to the wired-in definition.
If a primitive function is redefined, we simply delete the offending redefinition (with a warning). We use the scope rules for modules and conditionals to determine what the defining occurrence is. If a function name has no defining occurrence, we add the new defining occurrence of bottom bar.
- 1b) Rename all the defining occurrences so that each one is unique.
This is easy to do by adding a crosshatch and a number.
- 1c) Change all using occurrences of function names so that they have the same name as the defining occurrences.
- 1d) Change the scope of all function names so that they are all global.

Step 2: Eliminate complete language abbreviations by replacing them with their proper language counterpart.

Step 3: Put all the definitions in the (what is now one big) definition list into normal form. During this process, any IO functions or disk operations are treated as bottom bar. Ie, during this process, $f : x \rightarrow \text{error}$ for any such IO function f (where $\rightarrow$ denotes rewriting) and $\text{error} \rightarrow \text{error}$ and error has meaning bottom. If this process does not terminate, then the meaning of the expression (which is depending on this definition list) is bottom.

Those first three steps are all thought to occur "at compile time". Next comes "run time".

Step 4: Rewrite the expression.

During the rewriting process we have the rewrite rules for the primitive functions and the definition list provides us with a new set of rewrite rules for the other function names.

- if xq , then move element to nf, going left to right
- $p \Rightarrow e_1; e_2$ 1) reduce exprs of p to nf
working left to right
2) reduce $e_1 + e_2$ to normal form
- $e_1; e_2$ put e_1 in nf, be sure the result is a code
put e_2 in nf

$$\text{if } e_1 \text{ is } p \Rightarrow e'[v_1, \dots, v_n]; e'' \text{ w/ } f_0 + d_i \vec{d}_i$$

~~cond:~~

$$\text{cond}^* : \langle f_0 : e_2, c[d_i : e_2 / v_i], e'' \rangle$$

$$\text{condapp}' : \langle f_0 : e_2, c[v_1 \dots v_n], e'', \vec{d}_i, e_2 \rangle$$

$$\text{where } \text{condapp}' : \langle \text{"non } \nabla", e[v_1 \dots v_n] \dots \rangle$$

~~rewrite~~ $\rightarrow$

$$c[d_i : e_2 / v_i]$$

$$\text{and } \text{condapp}' : \langle \nabla, \sim, e'', \sim, e_2 \rangle$$

$$\rightarrow e'' : e_2$$

We give examples.

This is an example of how to write a composition operator that takes an arbitrary number of arguments. Just for fun we give two different ways of writing it. (It is interesting to note that these two definitions are equivalent even if the functions that we are composing do IO.)

```
def2 LongComp = ( null -> ~id ;
                  x. @ y. => x o ( LongComp : y ) ;
                  ~bottom ).
```

```
def2 LongComp = ( null => id ;
                  x. @. y. => ( LongComp : x ) o y ;
                  ~bottom ).
```

Our next example is how to write apply-to-all. Once again we give two equivalent ways (even in the presence of IO) of writing it.

```
def2 AppAll = ( f. =>
                ( null -> id ; al o [ f o 1 , ( AppAll : f ) o tl ] ) ;
                ~bottom ).
```

```
def2 AppAll = ( f. =>
                ( null -> id ; ar o [ ( AppAll : f ) o tlr , f o -1 ] ) ;
                ~bottom ).
```

In manipulating a list of animals sorted by weight, the programmer might write the following two patterns:

```
smallest .@ ( id @. largest. )
```

is the pattern that requires that the argument be a sequence of length at least 2 and defines smallest = 1, largest = -1.

```
( smallest. .@ id ) and ( id @. largest )
```

is a pattern that makes the same definitions as the previous pattern but now only requires that the sequence be of length at least one.

Data Type short example:

Outside:

```
delta;stack o push o e[ id , delta;stack ] = push
pop o delta;stack = pop
```

Axioms:

```
pop o push = 1 <-- e[ id , delta;stack ]
```

Implementation:

```
LIKE stack IS seq
```

```
def pop = 1 o StackToSeq
```

```
def push = SeqToStack o al o [ 1 , StackToSeq o 2 ] o (delta:len2)
```

EXPR.WHY

We explain some of the choices we made in defining expressions.

Patterns are not closed under the operations not, or.

=====

Basically, we made this choice since the definition list no longer makes sense. We first consider the operation "not". In this case, clearly the definition were made on the assumption that the argument had a certain form, and we have just negated that. For example, consider the alleged pattern not (x. @ y.). In this case, the pattern passes the argument iff the argument is not a sequence of length at least one. In this instance the definitions of x and y are totally meaningless. In the case of or, for example in the pattern p1 or p2, we don't know which of the patterns passed the argument so we don't know (at compile time) which of the patterns has meaningful definitions. We could rectify this by putting in runtime checks but that seems like a complication of patterns that is not worth the benefits. Finally, we note that the programmer can achieve the filter effects of not and or just by writing mere predicates without using the definitional characteristics of patterns.

Infix notation is introduced for => conditionals.

=====

In the proper language, there is no infix notation for one kind of conditional but not the other. The basic reason is that the kind of conditional included in the proper language is essentially lambda and so there is no easy way of expressing it in the rest of the language.

No also clauses are allowed in the => conditional.

=====

This decision was made since there were two possible interpretations of an also clause inside a => conditional and we couldn't decide which it should be. If we can't decide, then there is no natural choice and hence it might indicate that this use of the also clause would tend to lead to programmer errors. Basically there are two choices. Assume for example, we are dealing with the conditional

(e1 x. , y.] also h = e1(x,y) => e2(x,y,h) ; e3).

(In this example, we are even ignoring the case where h is recursive.)

- 1) Take also clauses to mean the same as they do for -> conditional.

In this case, the value of the expression applied to t would be (assuming that it passes the sieve) would be e2(x:t,y:t,h) where def x = 1 def y = 2 and def h = e1(x,y). or more succinctly, e2(1:t,2:t,e1(1,2)).

- 2) Take the also clauses to be more in the lambda spirit of =>.

In this case, the value of the expression applied to t would be (assuming that it passes the sieve) would be e2(1:t,2:t,e1(1:t,2:t)).

No labels on sequences for the proper language.

=====

The only way for a programmer to create a sequence labelled by an atom in the proper language is for him to use an operation like SeqToX inside of a data type construction. So labelled sequences DO exist in the semantic domain but the do NOT exist in the proper language. Note that this does not exclude labelled constructions from the complete language.

E-construction is sieve oriented.

=====

By its very nature, e-construction is sieve oriented. This is because if an e-construction does not see a sequence or sees a sequence of the wrong length, it returns del and does NOT return bottom (recall bottom indicates error). Since it is already sieve oriented, it seems right to make it sieve

if the answer for any of the components is del. This means that e-construction
does preserve sieves.

965 - IBM - 06