Chapter 3 - Language Definition

This section (semi-)formally describes the language. See Appendix 1 (Conventions) to see a description of what type face is used where.

Definitions

ReWrite supports only lists and 32-bit integers internally. However, for purposes of documentation the following types will be used:

integer (`int`) 32-bit integer

boolean (`bool`) either `true` or `false`

char (`char`) a character, represented as `"char "`

list (`list`) a list of zero or more values, denoted `val , .. ,val`
Note: this is represented as `lis` in ReWrite itself.

token-string (`sym`)
a token-string, represented `` `token-string ``

null (`null`) this is a data-less type, similar to `void` in C.
Note: this has no type representation in ReWrite - it is just a constant.

value (`val`)

a list or other atomic type. Basically anything.
This is just used for documentation purposes - it is not an actual type.

In addition other structures will be used (`set` etc). they will be defined as needed.

A trailing `-s` is used to indicate a sequence of zero or more of these. For example, `ints` is the same as `int , .. ,int ,` and represents a list containing one or more integers.

(directly coded) -
a hand coded function. These have been written directly in the intermediate language so as to run fast.

ReWrite Syntax

For the following, a token is considered to be any sequence of letters and numbers, starting with a letter. (an underscore is a letter). Note that ReWrite is case sensitive.

rules

A rewrite rule is something of one of the following forms:

```
name [patterns ] -> results ;
name [patterns ]::condition -> results ;
```

where

`name`	is a token,
`patterns`	is zero or more `patterns` , separated by commas,
`condition`	is an `expression` ,
`results`	is zero or more `expressions` , separated by commas.

patterns

A `pattern` is one of the following:

- `n` where `n` is a `constant` . This will match with the constant `n` . (see below)
- `x` where `x` is a token.

This will match with any one value.

`x` will be given this value for the duration of the rule.

- `x :type`

where `type` is one of `int` , `lis` , `bool` , `char` or `sym` (see below).

This will match with any one value of type `type` .

`x` will be given this value for the duration of the rule.

- `n :type`

a constant coerced to that type.

- `pp`

where `pp` is zero or more `patterns` , separated by commas.

The will match with a list, where the patterns pp must match the inside of the list.

- `.x`

where `x` is a token.

This will match with zero or more values.

`x` will be given a list containing these values for the duration of the rule.
• Any of the tokens`x` given above may be replaced by an `_` (underscore).
This matches with anything that a token would.
Multiple `_` in a pattern may refer to different objects.

types

In the `type` field above,
`int` specifies an integer,
`lis` specifies a list,
`bool` specifies a boolean (either `true` or `false`),
`char` specifies a character,
`sym` specifies a symbol (henceforth called a token-string).

A token-string will be a constant fixed by the system for each 'string'. For example, if I was to use the symbol `` `x ``, I could be sure that is was equal to itself, and different to anything else, and it only uses up as much storage as an integer for each use after the first. This would allow manipulation of expressions.

expressions

An `expression` is any of the following:
• `n`

where `n` is a `constant` .
• `x`

where `x` is a token.
`f [exprs ]`
where `f` is a token (a function name), and

`exprs` is a list or zero or more `expressions` separated by commas.
• `expression :type` is an expression coerced to the appropriate type.

coercions allowed:
`integer <-> character`
`integer <-> token-string` (this is provided for the compiler, and should not be used).

Note: `f [exprs ]` includes all the infix functions, and short forms, since these are just shorthand for `f [exprs ]` so:

```
.x
1+g[2]
0,.b
```
are all `expressions` .

constants

A constant may be:
• an `integer constant`,
• a boolean:
  - either `true` or `false`,
• a null:
  - `null`,
• a character:
  - represented `"char "`, for example `"a"`,
• a token-string:
  - represented `` `token-string `` , for example `` `thistoken ``,

An `integer constant` is one of the following:
•
a sequence of digits,
• `$`
followed by a sequence of hexadecimal digits,
• `%`
followed by a series of binary digits,
• `-`
followed by any of the above,
• `maxint`,
• `minint`.

Other shorthand forms and syntax:
`"string "`
allows several characters to be typed together.

For example:

```
"abc"
```

is equivalent to

```
"a","b","c"
```

```
'string '
```

is shorthand for typing in the ASCII values for the string entries. (This is shortly to be removed, and only remains because the compiler uses it).

For example:

```
'abc'
```

is equivalent to

```
97,98,99
```

```
(* comment  *)
```
is how comments are done.

Note that comments are fully recursive.

ReWrite Semantics

Recall that rewrite calls are of the form:

```
name [patterns ] -> results ;
name [patterns ]::condition -> results ;
```

This is how rules can be thought of as being evaluated:
• If an expression `name [values ]` is encountered during execution of code, a rule starting with the same `name` is searched for in the order last file in project to first file in project then top of file to bottom of file. For example, if the project consisted of two files, `setup` and `go` in that order, the search will (after first checking the internal libraries) start at the top of `go` and finish at the bottom of `setup`.
• As each rule is found, the values are matched with the patterns and if that succeeds then the condition (if there is one) is checked, and on success if that, the right hand side is evaluated. If either if these checks fail, then the search for the correct rule continues. If no matching rule is found, then an error is generated, giving the name of the rule that failed and the arguments that were passed to it.

Note: when Go is selected in the Project menu, then execution always starts with the expression

```
top[]
```

Precedence of operations

(similar to C apart from the not (!) operator)

```
()   []    (list)
```

```
: (type coercion)
. (splice)
*  /  %
+  -
<  <=  >  >=
=  !=
!
&
|  (or)
```

Some features

The above syntax results in some features that are worth pointing out:

• Functions can have a variable number of arguments.
Because the arguments passed to a function can be considered a list, the splice notation can be used for picking up parts of it.

For example,

```
second[a,b,.x] -> b;
```

returns the second argument passed,

```
last[.x,a] -> a;
```

returns the last argument passed.

• Functions can return an arbitrary number of results, including zero.

For example,

```
g[0] -> ;
g[n:int] -> g[n-1],n;
```

will return the results `1,2,3, ... ,n` not in a list.

If these results were wanted in a list then just use the list brackets, so

```
g[10] -> 1,2,3,4,5,6,7,8,9,10
```

Also, the case of returning nothing is not the same as returning `null` - the `null` is still a type (albeit one that contains no information), whereas returning nothing is just returning zero results.

• Functions can 'pick up' a variable number of arguments.

For example, (with `g` defined as above)

```
add[1000,2000] -> 3000;
add[1000,g[0],2000]
```

```
    -> add[1000,2000]
    -> 3000;
  add[1000,g[10],2000]
    -> add[1000,1,2,3,4,5,6,7,8,9,10,2000]
    -> 3055;
```

If `h` was defined as

```
  h[x] -> g[x];
```

the same result could be achieved by

```
  add[1000,.h[10],2000]
    -> add[1000,.1,2,3,4,5,6,7,8,9,10,2000]
    -> add[1000,1,2,3,4,5,6,7,8,9,10,2000]
    -> 3055;
```

These features make it unnecessary to actually use lists unless there is information that needs to be grouped (although lists can make code clearer), and gives extra flexibility to the language.

Note that `list` () and `splice` (`.`) are very powerful for getting the right level of list brackets.