Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Laboratório de Linguagens de Programação

SCRIPT 2.0
An Object Oriented Language for Denotational Semantics

by

Roberto da Silva Bigonha

LLP 005/99

Caixa Postal, 702
30.161-970 – Belo Horizonte
Minas Gerais – Brazil

may, 1999

Contents

Chapter 1

The Semantics Definition Language *SCRIPT*

1.1 Introduction

SCRIPT is a functional language aimed to provide a well-suited notation for conveying denotational semantic descriptions of programming languages in a structured fashion. To that end, issues such as description modularization, structural type equivalence, control of visibility, encapsulation, information hiding, inheritance and dynamic binding have been incorporated in the language's structure.

SCRIPT is a special purpose object oriented computer-processable language so that denotational semantic descriptions can be effectively *executed* and *debugged* with the help of computers.

Mosses' SSL and DSL notations[3, 9, 10] have been taken as *SCRIPT*'s starting point in the sense that many of its features have been adopted, notably grammar and abstract syntax specification notation, tuples, lists, parse tree nodes, patterns, CASE and LET notations. Other features such as list comprehension came from widely known functional languages Miranda[18] and ML[1].

A *SCRIPT* program is formed by three kinds of modules. There is a special module called `PROJECT`, which basically describes the environment in which *SCRIPT* programs are processed. The environment includes the idenfication of the main function of a formal definition, the input and output files associated with the main function domain, and the modules that compose an entire formal definition. There can exist only one `PROJECT` module per definition.

Another distinguised module is the `SYNTAX` module, which specifies the concrete and abstract syntax of a language. The syntactical elements of a language, whose syntax is defined by sections `SYNTAX` and `LEXIS` of this kind of module, serve to generated translators to render programs in the defined language into a λ-calculus notation for abstract syntax. There can exist only one `SYNTAX` modules per definition.

The third type of modules, simply referred to as `MODULE`, serves to encapsulate domain

and function definitions. These modules consist of the following optional sections:

- `EXPORTS`: identifies the entities exported by the module;
- `IMPORTS`: specifies the entities imported by the module;
- `DOMAINS`: declares variables and defines domains;
- `DEFINITIONS`: defines functions and other values.

Each $\mathcal{SCRIPT}$ module (`PROJECT, SYNTAX` or `MODULE`) can be separately compiled into an enriched version of λ-calculus, and ultimately linked together to form a machine processable denotational definition.

1.2 Basic Symbols

The reserved keywords of $\mathcal{SCRIPT}$ are:

```
AND       AUG       CASE      CAT     CONC      COMPONENTS
DEF       DIV       DOMAINS   EL      END       EQ
EXPORTS   EXT       FF                FOR       GE
GT        HEAD      IMPORTS   IN      INFILES   IS
LAM       LE        LET       LEXIS   LT
          MINUS     MODULE    MULT    N         NE
NOT       NUMBER    OR        OUT     OUTFILE   PRE
PLUS      PROJECT   Q         QUOTE   REM       RENAMES
SIZE      SYNTAX    T         TAIL    THIS      TRUTH
TT        UNIT      VAL
```

The special symbols are:

```
.  "  {  }  =  ^  ,  $  <  >  ;  '  |  +  -  /  *  !  :  [  ]  ?  (  )
```

1.3 $\mathcal{SCRIPT}$ Comments

Comments in $\mathcal{SCRIPT}$ descriptions start either with the symbol "`!`" or with "`--`", and end with the next *line-feed* or *form-feed* ASCII characters.

The pair of symbols "`--`" always iniciates a comment regardless where it occurs. This rules precludes the use of adjacent dashes within identifiers, although single dashes are allowed.

1.4 Syntax Summary

1.4.1 *SCRIPT* Modules

```
script        ::= module+ ;

module        ::= "PROJECT" project-ide pro-section* "END" project-ide ;
                | "SYNTAX"  syntax-ide  syn-section  "END" syntax-ide ;
                | "MODULE"  module-ide  mod-section* "END" module-ide ;

project-ide   ::= proper-noun ;

pro-section   ::= imports | domains | infiles | outfile | components ;

syntax-ide    ::= proper-noun ;

syn-section   ::= syntax domains lexis ;

module-ide    ::= proper-noun ;

mod-section   ::= exports | imports | domains | definitions ;
```

1.4.2 Project Section

```
infiles       ::= "INFILES" file-defn+ ;

outfile       ::= "OUTFILE" file-defn ;

file-defn     ::= domain-ide "=" filename ;

filename      ::= quotation ;

components    ::= "COMPONENTS" filename+-"," ;
```

1.4.3 Exports Section

```
exports       ::= "EXPORTS" exported-item+-"," ;

exported-item ::= closed-domain | open-domain | var-ide
                | domain-ide "." var-ide ;

closed-domain ::= domain-ide ;
```

```
open-domain   ::= open-sign domain-ide ;

open-sign     ::= "*" ;
```

1.4.4 Imports Section

```
imports       ::= "IMPORTS" window+ ;

window        ::= module-ide "(" imported-item+-"," ")" ;

imported-item ::= item | new-item "RENAMES" item ;

item          ::= closed-domain | open-domain | var-ide ;

new-item      ::= domain-ide | var-ide ;
```

1.4.5 Domains Section

```
domains       ::= "DOMAINS" dom-decl+-";" ;

dom-decl      ::= meta-var+-"," ":" domain-ide "=" dom-exp
                | meta-var+-"," ":" domain-ide
                | meta-var+-"," ":" "=" dom-exp
                | domain-ide "=" dom-exp ;

meta-var      ::= common-noun | domain-ide "." fun-ide ;

domain-ide    ::= proper-noun | builtin-dom ;

var-ide       ::= common-noun digit* prime* rep-op* ;

builtin-dom   ::= "Q" | "T" | "N" ;

dom-exp       ::= dom-a+-"|" ;

dom-a         ::= dom-b "->" dom-a | dom-b ;

dom-b         ::= dom-of-tuple | dom-of-node | dom-of-list
                | domain-ide   | quotation   ;

dom-of-tuple  ::= basic-tuple | dom-of-tuple "EXT"  basic-tuple ;

basic-tuple   ::= "(" field*-"," ")" | tuple-ide ;
```

```
tuple-ide     ::= domain-ide ;

field         ::= field-ide ":" dom-exp | field-ide | dom-exp ;

field-ide     ::= common_noun digit* prime* rep-op* ;

dom-of-node   ::= "[" dom-c* "]" ;

dom-c         ::= domain-ide rep-op* | quotation ;

dom-of-list   ::= dom-b rep-op ;
```

1.4.6 Syntax Section

```
syntax        ::= prod-range+-";" ;

prod-range    ::= production | range ;

production    ::= nonterminal "::=" alternative+-"|" ;

range         ::= nonterminal "===" spec+-"|"
                | nonterminal "=/=" spec+-"|" ;

lexis         ::= "LEXIS" unit-def prod-range+-";" ;

unit-def      ::= "UNIT" "::=" nonterminal+-"|" ;

alternative   ::= element* ":" exp-b | element* ;

element       ::= symbol sep-op terminal | symbol rep-op | symbol ;

symbol        ::= nonterminal | terminal ;

nonterminal   ::= common-noun ;

terminal      ::= quotation ;

sep-op        === "*-" | "+-" ;

spec          ::= terminal | one-char-str ".." one-char-str ;
```

```
one-char-str  ::= quotation ;
```

1.4.7 Definitions Section

```
definitions   ::= "DEFINITIONS" def-binding* ;

def-binding   ::= "DEF" definition ;

definition    ::= lhs "=" exp ;

lhs           ::= pattern-exp
                | fun-head
                | fun-head pattern-exp+
                | fun-head pattern-exp+ ":" dom-exp ;

fun-head      ::= fun-ide | domain-ide "." fun-ide ;

fun-ide       ::= common-noun digit* prime*;

pattern-exp   ::= pattern-exp pat-di-op pattern-a | pattern-a ;

pattern-a     ::= pat-mon-op pattern-a | pattern-b ;

pattern-b     ::= "(" pattern-exp*-"," ")" | "[" pattern-c* "]" | "<" ">"
                | pattern-c ;

pattern-c     ::= var-ide | var-ide ":" dom-b | literal-const ;

pat-di-op     ::= "PRE" | "CAT" ;

pat-mon-op    ::= "NUMBER" | "QUOTE" | "TRUTH" | "VAL" ;

exp           ::= "LAM" pattern-exp "." exp
                | let-binding+ "IN" exp
                | exp-a "->" exp else-symbol exp
                | exp-a updating-exp+
                | exp-a seq-op exp
                | exp-a ;

let-binding   ::= "LET" definition ;

else-symbol   ::= "ELSE" | "," ;
```

```
exp-a          ::= exp-b "IS" pattern-exp | exp-b ;

exp-b          ::= exp-b di-op exp-c | exp-c

exp-c          ::= mon-op exp-c | exp-d ;

exp-d          ::= exp-d exp-e | exp-e ;

exp-e          ::= case-exp | tuple-exp | list-exp | node-exp | exp-f ;

case-exp       ::= "CASE" exp-a clause+ "END"

clause         ::= "/" pattern-exp+-"/" "->" exp ;

list-exp       ::= "<" exp*-"," ">" | "<" exp-b ".." exp-b ">"
                   | "<" exp "|" qualifier+-"|" ">" ;

qualifier      ::= generator | generator "::" filter ;

generator      ::= pattern-exp "<-" source-exp ;

source-exp     ::= exp ;

filter         ::= exp ;

tuple-exp      ::= "(" exp*-"," ")" ;

node-exp       ::= "[" exp-g* "]" ;

exp-f          ::= exp-g | father_fun | field_qualif ;

exp-g          ::= literal-const | variable ;

variable       ::= var-ide | "THIS"  ;

father_fun-    ::= fun-ide "^" ;

field-qualif   ::= object-exp "." field-ide+-"."

object-exp     ::= variable | "(" exp ")" ;

updating-exp   ::= "{" binding-pair+-"," "}" | "{" exp "}" ;
```

```
binding-pair  ::= exp "=" exp ;

seq-op        ::= ";" | "$" ;
```

1.4.8 Miscellaneous

```
uppercase     ::= "A" .. "Z" ;

lowercase     ::= "a" .. "z" ;

anycase       ::= lowercase | uppercase | "-" ;

proper-noun   ::= uppercase anycase* ;

common-noun   ::= lowercase anycase* ;

digit         ::= "0" .. "9" ;

prime         ::= "'" ;

rep-op        ::= "*" | "+" ;

di-op         ::= "AND"  | "OR"  | "EQ"   | "NE"   | "LT"   | "GT"
                | "LE"   | "GE"  | "PLUS" | "MULT" | "DIV"  | "REM"
                | "CAT"  | "AUG" | "PRE"  | "EL"   ;

mon-op        ::= "NOT"  | "NUMBER" | "QUOTE" | "TRUTH" | "CONC"
                | "SIZE" | "VAL"    | "NEG"   | "OUT"   | "HEAD"
                | "TAIL" ;

literal-const ::= number   | quotation | "TT" | "FF" | | "?" ;

number        ::= digit+ | "-" digit+ ;

quotation     ::= "\"" quotation-ch* "\"" ;

quotation-ch  ::= char-1 | special-char ;

char-1        =/= "\\"

special-char  ::= "\b" | "\\" | "\ddd" | "\r" | "\f" | "\t"
                | "\n" | "\0" | "\"";
```

Chapter 2

Syntax Specification

2.1 Introduction

The purpose of the syntax specification module is to provide a notation for specifying concrete syntax of a programming programming language and for indicating how the abstract syntax parse tree for programs in the language can be derived from a given concrete syntax. The syntax module is composed of three sections named `SYNTAX`, `DOMAINS` and `LEXIS`.

2.2 Syntax Specification

The `SYNTAX` section consists of a set of production rules of a context free grammar. Each production has a nonterminal symbol to the left of "`::=`", and a list of alternatives, separated by "`|`", to the right.

The first production occuring in the `SYNTAX` section defines the grammar starting symbol. Nonterminal symbols of the grammar are formed from lower-case letters and dashes. A terrminal symbol of the grammar is a quotation, which consists of a sequence of characters written in quotes (`"`).

A list of repeated terminal or nonterminal symbols can be specified as an iterator, which can written as:

- `x*` = zero or more occurences of grammar symbol `x`,
- `x+` = one or more occurences of grammar symbol `x`;
- `x+-t` = zero or more occurences of grammar symbol `x`, separated by terminal `t`.
- `x+-t` = one more occurences of grammar symbol `x`, separated by terminal `t`.

A production alternative defines a possibly empty sequence of terminals, nonterminals or iterators. For example, the classical grammar for expression can be specified as:

```
exp    ::= exp "+" term
         | term ;
term   ::= term "*" factor
```

```
          | factor ;
factor ::= ide | constant | "(" exp ")" ;
```

There is also a special kind of production rules called *range*, whose syntax is borrowed from [10]. Ranges are production rules whose alternatives can only be a single terminal symbol or an interval of ASCII characters. The value produced is always the specified terminal symbol recognized as a quotation. Ranges are distinguised from normal production rules by the use of the symbols "`===`" or "`=/=`" to separate the production sides instead of "`::=`". The "`===`" symbol defines the left hand side as the set of terminal specified in the alternatives and the "`=/=`" symbol defines it as the complement of the specified set.

In the example

```
digit        === "0" .. "9" ;
comment-char =\= ";" ;
```

the range defined by `digit` above is equivalent to:

```
digit === "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

and `comment-char` is anything but a semicolon.

2.3 Domains and Abstract Syntax

From the concrete syntax specification, the *SCRIPT* compiler generates parse-tables, scanner routines and ultimately produces a compiler that translates programs in the specified language into abstract syntax tree code. Normallly, the abstract syntax tree is generated during the program parsing by creating nodes corresponding to productions involved in the recognition process. Unless specified otherwise, nodes are constructed from the values associated with each grammar symbols occurring in the production alternative. The value associated with terminal symbols is the terminal itself, i.e., a quotation. The value associated with nonterminal is the value produced when the nonterminal was reduced by the parser. The label of each node is the concatenation of the domain names of the grammar symbols occuring in the corresponding production alternative. The domain of terminal is the quotation itself. The domain of noterminals can be inferred by default, by capitalizing its first letter, or it is explicitly declared in a `DOMAINS` section. The resulting node becomes the value associated with the nonterminal on the left hand side of the production rule.

However, if a different value is desired, a value-expression can be attached to the production alternative. In this case, the value produced is that of the attached expression, which, in principle, can be an expression of any type. A value-expression is any valid *SCRIPT* expression whose operands are terminal or nonterminal symbols occurring in the corresponding production altenative. The value of each operand is that of the value expression implictly or explicitly associated with it. Repeated nonterminals should be referenced in the same order they appear in the production alternative so as to avoid ambiguity.

This facility for producing abstract syntax tree allows the following type of transformations:

1. Elimination of precedence information present in concrete syntax specifications by an appropriate domain specification.
2. Placement of constructs with similar semantic properties in a single syntactic domain, while making sure that domains with different semantic roles are distinct.
3. Elimination of production rules which lose their semantic significance after the operations described in items above are performed. These are, in general, productions that would only yield extra "chain-reduction" nodes in the parse tree.
4. Elimination and addition of delimiters (terminal symbols) to simplify and unify the structure of constructs.
5. Change of the order of occurrence of the constituents of constructs. For example, if a+b, +ab and ab+ are different representations of the same expression, it might be desirable to reorder the components of some of them in order to have just one abstract form. Operations of this type are useful to reduce the number of semantic equations.

For example, given the production rule:

```
exp ::= exp "+" factor  |  factor ;
```

where `exp` and `factor` are in the domain `Exp`, the resulting abstract syntax production would be:

```
Exp   = [ Exp "+" Exp ]  | [ Exp ];
```

Nodes which correspond to alternatives that contain only one nonterminal symbol, such as `[Exp]` above, can be eliminated by making that nonterminal symbol itself the corresponding value-expression, as in:

```
exp ::= exp "+" factor
      | factor : exp;
```

It is also possible to delete or add terminal symbols to abstract syntax alternatives even if they do not exist in the concrete specification. The example that follows illustrates this system capability. Suppose we have the following productions and value-expressions:

```
SYNTAX
    s-exp ::= s-exp s-exp-seg : [ "(" s-exp "." s-exp-seg ")" ];
    func  ::= "label" "[" ide ";" func "]" : [ "label" ide ";" func ] ;
```

and the domain specification

```
DOMAINS
    s-exp, s-exp-seg : S;
    func             : F;
    ide              : Ide;
```

The resulting abstract syntax is:

```
    S  = [ "(" S "." S ")" ]
    F  = [ "label" Ide ";" F]
```

2.4 Lexical Specification

The LEXIS section of a syntax module always starts with a fixed production rule of the form:

```
unit-def ::= "UNIT" "::=" nonterminal+-"|"
```

where each production alternative specifies the units which are to be recognized by a parser associated with LEXIS. The left part of this production rule is always the keyword UNIT, whereas the right part is as usual, except that the attached value-expressions must be tuples, each denoting the token value to be returned by the generated scanner.

All quotations ocurring in the SYNTAX section cause an implicit addition of a suitable alternative to the definition of UNIT, so all terminal cited in the grammar can be properly recognized. This automatic inclusion is not performed if the quotation is marked by keyword OUT.

2.5 A Small Example

```
SYNTAX MiniL

   prog ::= "program" body "end" ;
   body ::= "read" id ";" cmd+-";" ";" "write" exp
              : ["read" id cmd+ "write" exp];
   cmd  ::= id ":=" exp | cmds
         | "while" exp "do" cmds "end" ;
   cmds ::= cmd+-";" : cmd+ ;
   exp  ::= "ID" id  | "suc" exp | "NM" num;

LEXIS

   UNIT   ::= id   : (OUT "ID", id) ;
            | num : (OUT "NM", num) ;
   id     ::= letter+ : QUOTE letter+;
   letter === "a" .. "z" ;
   num    ::= digit+ : NUMBER digit+;
   digit  === "0" .. "9" ;

END MiniL
```

Chapter 3

Domains

3.1 Introduction

SCRIPT domains are complete partial orders with a minimal element *bottom* [11, 12, 13, 14, 15, 16, 17, 19, 20]. Domains have properties to guarantee that solutions of possibly reflexive domain equations always exist up to isomorphism. The special value *bottom*, which is not directly representable in *SCRIPT* y, serves to model the semantics of non-termination. The sigleton domain "?" contains the special *undefined* value, which is also represented as "?", and is used to indicate the value of semantically *non sensical* expressions.

3.2 Domain Identifiers

A domain name consists of a sequence of one or more letters, possibly containing embedded hyphens (`"-"`), and that always starts with a capital letter, e.g., `Store, Command, Environment`. Domain names denote built-in standard domains or user defined domains.

3.3 Standard Domains

Domains `N, Q, T` and `?` are standard, and thus directly available in every semantic definition, and each has a number of pre-defined operations. In these operations, if any of the operands is *bottom*, the result of the operation *bottom* respectively, otherwise it denotes the expected value.

3.3.1 Domain of Integers

`N` is the flat domain [16] of integer numbers. The defined operations on members `n1` and `n2` of `N` are listed below.

- `n1 PLUS n2` - add

- `n1 MINUS n2` - subtract
- `n1 MULT n2` - multiply
- `n1 DIV n2` - divide
- `n1 REM n2` - remainder
- `NEG n1` - change sign
- `n1 LT n2` - less
- `n1 LE n2` - less or equal
- `n1 GT n2` - greater
- `n1 GE n2` - greater or equal
- `n1 EQ n2` - compare equal
- `n1 NE n2` - compare not equal

Constants in domain `N` are the integer decimal numbers in the range `-38,768..32,767`. Negative numbers are represented as usual, e.g. -3000.

3.3.2 Domain of Quotations

`Q` is the flat domain of quotations or strings. If `q`, `q1` and `q2` are in domain `Q`, and `q*` in domain `Q*` of list of quotations, the following operations are defined:

- `q1 LT q2` - less in lexographic order
- `q1 LE q2` - less or equal
- `q1 GT q2` - greater
- `q1 GE q2` - greater or equal
- `q1 EQ q2` - compare equal
- `q1 NE q2` - compare not equal
- `q1 CAT q2` - the quotation formed by the concatenation of quotations `q1` and `q2`.
- `QUOTE q*` - the quotation whose characters are the concatenation of the elements of list `q*`. For instance, the expression `QUOTE <"This", " is", " ", "it">` denotes quotation `"This is it"`.
- `NUMBER q*` - the decimal number whose digits are the components of `q*`, which must have at least one component. Otherwise it is the value undefined.
- `TRUTH q*` - the value `TT` if `q*` is `<"T","T">`; the value `FF` if `q*` is `<"F","F">`; otherwise it is undefined.

The allowed constants in domain `Q` are strings or quotations, which are sequences of ASCII characters enclosed in quotes ("), e.g., `"This is a quotation!"`.

Special characters are entered in quotations as shown below:

Name	Coded as	Name	Coded as	Name	Coded as
backspace	`\b`	carriage return	`\r`	newlines	`\n`
backslash	`\\`	form feed	`\f`	null character	`\0`
bit pattern	\ddd[1]	horizontal tab	`\t`	quote	`\"`

For example,

```
"She said: \"This is a quotation\""
```

denotes the quoting of

```
She said: "This is a quotation".
```

3.3.3 Domain of Truth Values

`T` is the flat domain of truth-values. If `t1` and `t2` are expressions in the domain `T` of truth-values, then the following operations are valid:

- `t1 AND t2` - logical and
- `t1 OR t2` - logical inclusive or
- `NOT t1` - logical negation
- `t1 EQ t2` - compare equal
- `t1 NE t2` - compare not equal

The constants in domain `T` are `TT` and `FF`, for **true** and **false**, respectively.

3.3.4 Domain of Undefined Values

The flat domain of undefined values is represented by the symbol `?`. Any $\mathcal{SCRIPT}$ operator can be applied to undefined value `?`; the result is always undefined. Functions can also be applied to `?`, but the result depends on the evaluation of the function's body, because $\mathcal{SCRIPT}$ implements the *lazy* mechanism for parameter passing [4, 8]. Any function can also return the undefined value.

3.4 Constant Domains

All quotations are in domain `Q`. However, for technical reasons, any quotation occurring in places where a domain is expected is considered to represent the domain whose only proper non-bottom element is the quotation itself. The name of this domain is the quotation itself. For instance:

```
DOMAINS
    Mode = "int" ;
```

The string `"int"` when occurring in domain expressions denotes a domain containing only the quotation `"int"`.

3.5 Variable Declaration

An identifier denoting a variable, i.e., a function, a field name or other values, consists of a sequence of one or more letters, possibly containing embedded hyphens ("-"), and

that always starts with a lower-case letter. Variable identifiers may also be "decorated" (suffixed) by sequence of decimal digits and/or primes, and optionally ended by a sequence of `"*"` and/or `"+"` signs to suggest that they are lists. The following are valid *SCRIPT* variables: `r-value, r-value*, r-value1, r-value1*, r-value', r-value1''`.

Variable declarations serve the purpose of associating undecorated variables with their domains, and, if necessary, to provide *denotations* for new user defined domains. A declaration has the following general format:

variable names `:` *domain name* `=` *domain expression* `;`

whose parts may be omitted in a particular definition as long as at least two of them are present. The delimiters (":","=",";") must be always present so that the declared elements can be easily identified by the compiler. For example, the following variable declarations illustrate of the possible cases:

```
DOMAINS
     a,b,c  :  A =  N -> N ;
     d,f,g  := (f1:N, a*, b:Q)
     B      =  (N) -> N ;
     x,y,z  :  (A) ;
```

Variables `a, b` and `c` above are in domain A, which is the domain of functions from `N` to `N`. Variables `d, f` and `g` are tuples of three components of type `N`, `A*` and `Q`. `B` is defined to be the domain of functions from tuples `(N)` to `N`. And finally, variables `x, y` and `z` are one component tuples containing a function from `N` to `N`.

All variables must have its type known before they are used. However, not all variables need be explicitly declared. Moreover, decorated variables can not be explicitly declared in a domain section. Their domains are implicitly the domains of their undecorated versions. On the other hand, in the context of function or parameter declaration, decorated variable that represents a formal parameter may be declared.

Unless a variable is explicitly declared, *SCRIPT* adopted the following convention:

- Any undecorated variable is assumed to be in the domain whose name is that of the variable with the first letter capitalized.
- Any variable decorated with primes, decimal digits is implicitly in the same domain as its corresponding undecorated version. For example, undeclared variables `s, s1` and `s'` are by default in the domain `S`.
- if `a` is in domain `A`, then the occurrences of `a+` and `a*` are interpreted as member of `A+` and `A*` respectively.

These conventions are intended to contribute to description compactness and convenience of writing. Since they are very simple and uniform, no loss of readability is expected.

Variables may be declared in a `DOMAINS` sections or at the binding points. For instance,

```
DEF f ( n ) : Q = ...
DEF f   n1  : Q = ...
```

declares two functions, both named `f`: one, in the domain `(N) -> Q`, maps tuples whose single component is in `N` to `Q`, and the other is in the domain `N -> Q`. The domains of the function arguments are used to resolve the name overloading.

3.6 User Defined Domains

There exists a variety of domain operators so that the user can create domains expressions denoting more complex domains to model syntactic or semantic properties of programming languages.

Domain expressions normally occur in `DOMAINS` sections as the denotations of new domains, but they can also be attached to any variable name occurring in binding context in order to provide their domains locally. A domain expression may be a domain name, a quotation, which is assumed to denote a domain whose only proper element is that constant, or a combination of simpler domain expressions and domain operators. Particularly, a domain expression may denote union of domains, domain of tuples, domain of lists, domain of nodes or domain of functions.

In the following , assume that $d, d_1, \cdots, d_n$ are arbitrary domain expressions, $D_1, \cdots, D_n$ are domain identifiers or quotations, q is a quotation and $a_1, \cdots, a_n$ are variable identifiers.

The definition of a new domain may appear in any order. In fact, a domain expression may even contains names of domains which are not defined in the module scope. This should not cause any problem or error unless the undefined domains are effectively used in an expression in the module.

After processing each block of domain declarations (domain clauses), all domain definitions in the current module should be re-checked to remove any dependency on undefined domains which have been defined in the block.

3.6.1 Union of Domains

The domain expression d_1 `|` ... `|` d_n represents the **union** of the domains denoted by expressions $d_1, d_2, \ldots, d_n$.

$\mathcal{SCRIPT}$ uses plain *union* (`|`) as opposed to *separated sum* (+) of D. Scott [14]. This approach relieves the user of having to cope with *projections* and *injections* between a sum and its summands throughout denotational descriptions. So the domain of values need not to be carried at run-time. Note, however, that to ascertain from which operand of a union a given value come, it is necessary that the operands are distinguishable by the enquiry operation `IS`, which will be defined later.

Alternatively, a definition of a union can be unfolded into a sequence of simpler definition with the same left hand side. Distinct definitions of the same domain introduce a union definition. Both styles lead to equivalente domain definitions.

3.6.2 Domain of Tuples

The domain expression $(a_1 : d_1, \ldots, a_n : d_n)$ represents the domain of n-**tuples** whose i-th component is in the domain denoted by d_i, and can be selected by field identifier a_i, for $1 <= i <= n$. This is the cartesian product of domains. For any component, either its field name or its corresponding field domain may be omitted. If the domain of a field is not specified, the default domain is assumed. The domain denoted by the expression above is $(d'_1, \ldots, d'_n)$, where each d'_i, for $1 \leq i \leq n$, is either the domain d_i explicitly specified or assumed by default from the name of the field a_i.

Hierarchy of Domains

Domains of tuples are *extensible* in the sense that a tuple domain can be defined as an extension of another domain of tuples. No domains other than tuple domains are extensible, and thus classified as *non-extensible*.

The expression d_1 `EXT` d_2 represents the domain of tuples extended from the domain of tuples d_1, i.e., it denotes a domain of tuples whose elements are the concatenation of the elements of tuples in d_1 followed by the elements of tuples in d_2. The heading components of the extended tuple have the same field names as those of its base tuple, and field names cannot be repeated.

Indeed, an operation d_1 `EXT` d_2, where d_1 and d_2 are domains of tuples, defines the relation *is-a* between d_1 and d_2, i.e., operator `EXT` creates hierarchies of domains of tuples. Alternatively, it is possible to create a domain of extended tuples directly, that is, a given domain of members of some hierarchy, just by listing all their domain components. In the example below, domains `A-elem`, `B-elem` and `C-elem` are valid extensions of `Stk-elem`.

Given the domain declaration `A = B EXT` c, where `B` is a domain name and `c` denotes a domain of tuples, and `A` is not a union domain, then `A` is defined to be a *direct extension* of domain `B`, and `B` is a *direct base* of `A`. Thus, a domain A is defined to be an *extension* of a domain B if

1. A and B are the same domain identifier or
2. A is a direct extension of an extension of B.
3. The domains of the fields of B are the equivalent to domains of the heading fields of A, in the same order. The names of the fields are irrelevant.

Conversely, a domain B is a *base* of a domain A if A is an extension of B.

There are two operations defined on tuples: field selection and tuple construction.

Field Selection

Selection of fields of tuples is expressed via the dot notation t`.f`, where t is a tuple expression and `f` is one of its field names.

Tuple Construction

Tuples can be constructed by explicitly enumerating its components by means of the notation $(e_1, \cdots, e_n)$ where expressions e_i, for $1 \leq i \leq n$, define the values of the components. Note that pairs of parentheses are always an operator to build tuples, and so they must be used consciously.

A new tuple can also be created from another by redefining some of its fields. This kind of tuple is called *updating tuple* and is of the form $t\{f_1{=}v_1, \cdots, f_n{=}v_n\ \}$, where t and v_i are expressions, f_i field names, for $1 \leq i \leq n$. This operation creates a new tuple which has the same components of tuple resulting from the evaluation of expression t, except that the components identified by fields f_i contains the values of v_i.

It is also possible to create tuples by means of the binary operator `CAT`, which concatenates two tuples to construct bigger one.

```
DOMAINS
    Stk      = Stk-elem* ;
    Stk-elem = (a : N) ;
    A-elem   = Stk-elem EXT (b : N);
    B-elem   = A-elem   EXT (c : N, d : N) ;
    C-elem   = (N, N, N, N) ;
DEFINITIONS

DEF stk0      = <>

DEF stk-elem0 =(1)

DEF push(stk)(stk-elem) : Stk = stk-elem PRE stk

DEF mark(stk-elem)(n) : Stk-elem = stk-elem{a=n}

DEF teste : Stk =
    LET a-elem1 = stk-elem0 CAT (2)
    LET a-elem2 = mark(a-elem1)(100)
    LET b-elem1 = (11,12,13,14)
    LET c-elem1 = (21,22,23,24)
    LET stk1    = push(stk0)(a-elem2)
    LET stk2    = push(stk1)(b-elem1)
    LET stk3    = push(stk2)(c-elem1)
    IN  stk3
```

Tuple `a-elem1` has all the elements of `stk-elem0` followed to a element with value 2. Domain `Stk` is the domain of finite lists of components whose base type is `Stk-elem`. Note that the formal parameter of functions `push` and `mark` can correspond to any extension of

`Stk-elem`, which makes these functions truly polymorphic [6]. Note that the expression `stk-elem{a=n}` denotes a value constructed from that currently bound to `stk-elem`.

3.6.3 Domain of Lists

Expressions of the form d`+` denotes domains of **finite non-empty lists** whose components are in d. d`*` is the domain of possibly **empty finite list** whose components are in d. An instance of a list is denoted as `<`$a_1, a_2, \cdots, a_n$`>`, where each $a_i \in d$, for $1 \leq i \leq n$.

Assume that $e, e_1, \ldots, e_n$ are arbitrary $\mathcal{SCRIPT}$ expressions, and x a variable name. Lists can be built up and manipulated by means of the following operations: creation, indexing, length, concatenation and augmentation.

List Creation

The operations to create new lists are: list enumeration, list range, list updating and list comprehension.

List enumeration is an operation to create lists by enumerating their components by means of the notation `<` $e_1, \cdots, e_n$`>`, where all expressions e_i, for $1 \leq i \leq n$ are in the same domain. The domain of the list is the domain of the elements suffix by "`*`". The empty list is denoted as `<>`. The exacty domain of the empty list depends on the context it occurs.

List range is an operation to create lists of integer values or lists of character values via interval specification of the form e_1`..`e_2, where e_1 and e_2 are integer expressions. The interval specification denotes a list containing the sequence of all values in a given range. For instance, `10..12` represents the list `<10,11,12>`.

List comprehension is another notation to built lists. It employs a syntax adapted from conventional mathematics for describing sets. Its syntax is

`<`e `|` p_1 `<-` s_1 `::` f_1 `|` $\cdots$ `|` p_n `<-` s_n `::` f_n`>`

where e is a valid expression and, for $1 \leq i \leq n$, p_i is a pattern-expression (see section 4.3), s_i a list expression called *source list*, and f_i are optional logical expressions called *filters*. Terms of the form p_i `<-` s_i are called *generators*. The term p_i `<-` s_i `::` f_i is called a qualifier. A list comprehension may contain an arbitrary number of qualifiers, which are evaluated form left to right. The identifiers occurring in p_i are successively bound to elements drawn from the corresponding source list s_i and, together with identifiers bound in the previous qualifiers, may be used in e, s_k and f_k, for $k \geq i$.

The components of the list comprehension are computed as follows:

1. Component values are drawn in the order they occur on the source list of each generator.
2. Values produced by each generator are matched against the corresponding pattern. If all values match, identifiers encountered in the patterns are properly bound to parts of the drawn values (see section 4.3).
3. Then all filters are evaluated in the scope of the identifiers bound in the above step.
4. If all filters succeed, i.e., they all evaluate to `TT`, the component e is computed and inserted in the resulting list.

List updating is an operation to create new lists from another by redefining some of its elements. An updating list of the form

$$e\{i_1 = v_1, \cdots, i_n = v_n\}$$

where e is a list expression, v_j are expressions, i_j are integer expressions whose values satisfy the condition $i_j \leq$ `SIZE` e, for $1 \leq j \leq n$, denotes a new list, which has the same elements of list e, but with the values at the positions designated by i_j containing the values of v_j. If any $i_j >$ `SIZE` e, the resulting list is the undefined value.

List Length

The operation `SIZE` $e*$ gives the number of components of list e*.

List Indexing

The elements of a list can be retrieved through indexing operation of the form $e*$ `EL` k, where k is an integer expression. This operation produces the value of the k-th component of list `e*`, or has the undefined value ? if the value denoted by k is greater than `SIZE` $e*$ or less than 1.

List Concatenation

Binary concatenation of lists is of the form e_1* `CAT` e_2*, where e_1 and e_2 are lists in the same domain. The result of this operation is a list whose components are those of e_1* followed by those of e_2*.

Unary concatenation of lists is of the form `CONC` $e**$ where $e**$ is a list of lists. The result is a list formed by concatenating (`CAT`) the lists that are the components of $e**$. If $e**$ is in domain `A**`, the resulting list is in domain `A*`.

List Augmentation

The operation e `PRE` $e*$, where the value of e is in the same domain as the elements of $e*$, produces a new list whose head (first component) is e and whose remaining components (its tail) are those of $e*$.

Similarly, $e*$ `AUG` e is the list whose components are those of $e*$ appended with the value denoted by e.

List Decomposition

The unary operations `HEAD` $e*$ and `TAIL` $e*$ return the first and the remaining elements of a list $e*$, respectively.

3.6.4 Domain of Continuous Function

The domain expression d_1 `->` d_2 denotes the domain of **continuous functions** [14, 16] from d_1 to d_2. The operator "`->`" has precedence over "`|`" and associates to the right, for instance, d_1 `->` d_2 `->` d_3 is equivalent to d_1 `->` `A`, where is `A` is defined as d_2 `->` d_3.

3.6.5 Domain of Nodes

A domain expression of the form `[`$D_1 \ldots D_n$`]` represents the domain of **tree nodes**, which consist of a label and a tuple of emanating branches.

The operator "`[` $\cdots$ `]`" requires D_i, for $1 \leq i \leq n$, to be either quotation or domain identifier possibly followed by a sequence of `*`'s or `+`'s.

The label serves to distinguish nodes and is implicitly defined by the quotation:

`QUOTE <`$q_1, ..., q_n$`>`,

where each q_i, for $1 \leq i \leq n$, is the name of the domains occurring in the same order in `[`D_1,$\cdots$, D_n`]`. Quotations in domain expressions denotes constant domains.

The tuple of branches emanating from each node is formed with values in the non-quotation domains specified in `[`$D_1 \ldots D_n$`]`. A tree node can be viewed as a **labeled tuple** with no field names.

Members of the domain `[`$D_1 \cdots D_n$`]` are represented as `[`$e_1 \ldots e_n$`]`, where $e_1, \ldots, e_n$ are expressions in the domains $D_1, \ldots, D_n$, respectively.

3.7 Domain Equivalence

In $\mathcal{SCRIPT}$, types are synonym to domains. The type discipline is such that everything concerned with types is conducted at compile time so that once the type checker has accepted a $\mathcal{SCRIPT}$ definition, objects do not have to carry their types at run time. All new domains must always be explicitly declared in a `DOMAINS` section of some module. The type discipline of $\mathcal{SCRIPT}$ is based on **structural equivalence**[2], which is defined as follows.

Two domains A and B are **equivalent** if and only if at least one of the following applies:

1. A and B are identical domain names.

2. A and B are distinct domain names, each being the recursive reference to a reflexive domain definition occurring in exactly the same corresponding position in their domain structures.

3. A and B are the same constant domain of a single quotation.

4. A and B are the domain ? of undefined values.

5. A and B are distinct domain names whose visible definitions are, respectively, the longest chains $A = A_1 = A_2 = ... = A_n$, and $B = B_1 = B_2 = ... = B_m$, where A_i, for

$1 \leq i \leq n$, B_j, for $1 \leq j \leq m$, are domain names, and A_n and B_m are the first and only repeated names in their respective chains.

6. B is a domain expression and A is a domain name, whose visible definition is such that $A = d$ or $A = A_1 = A_2 = ... = A_n = d$, where A_i, for $1 \leq i \leq n$, are domain names, and domain expression d is equivalent to domain expression B.

7. A is a domain expression and B is a domain name whose visible definition is such that $B = d$ or $B = B_1 = B_2 = ... = B_n = d$, where B_i, for $1 \leq i \leq n$, are domain names, and domain expression A is equivalent to domain expression d.

8. A and B are domains of lists, A is of the form a`*`, B is of the form b`*`, and a is equivalent to b.

9. A and B are domains of lists, A is of the form a`+`, B is of the form b`+`, and a is equivalent to b.

10. A and B are both the polymorphic domain of empty lists.

11. A and B are domains of tuples, A is of the form $(a_1, \cdots, a_n)$, B is of the form $(b_1, \cdots, b_n)$, and for $1 \leq i \leq n$, a_i is equivalent to b_i. Field names are not relevant to determine domain equivalence. Bound functions are rightly considered tuples components.

12. A and B are unions of domains, A is of the form a_1 `|` a_2 `|` $\cdots$ `|`a_n, B is of the form b_1 `|` b_2 `|` $\cdots$ `|`b_n, and for $1 \leq i \leq n$, a_i is equivalent to b_i.

13. A and B are domains of nodes whose labels are identical.

14. A and B are domains of continuous functions, A is of the form a_1 `->` a_2, B is of the form b_1 `->` b_2, a_1 is equivalent to b_1, and a_2 is equivalent to b_2.

3.8 Domain Compatibility

The type checking discipline requires that variables can only occur in contexts where their domains are **compatible** to the domains expected for those contexts.

The notion of domain compatibility in $\mathcal{SCRIPT}$ merges the concepts of structural equivalence defined above and type inclusion [6] found in many imperative programming languages.

A domain A is **compatible** to a domain B if and only one of the following applies:

1. A and B are identical domain names.

2. A and B are distinct domain names, each being the recursive reference to a reflexive domain definition occurring in exactly the same corresponding position in their domain structure.

3. B is a domain expression, A is a domain name whose visible definition is such that $A = d$ or $A = A_1 = A_2 = ... = A_n = d$, where A_i, for $1 \leq i \leq n$, are domain names, and domain expression d is compatible to domain expression B.

4. A and B are distinct domain names whose visible definitions are, respectively, the longest chains $A = A_1 = A_2 = ... = A_n$, and $B = B_1 = B_2 = ... = B_m$, where A_i, for $1 \leq i \leq n$, B_j, for $1 \leq j \leq m$, are domain names, A_n and B_m are the first and only repeated names in their respecive chains.

5. A is a domain expression and B is a domain name whose visible definition is such that $B = d$ or $B = B_1 = B_2 = ... = B_n = d$, where B_i, for $1 \leq i \leq n$, are domain names, and domain expression A is compatible to domain expression d.

6. A and B are domains of lists, A is of the form a`+`, B is of the form b`*` or b`+` and a is compatible to b.

7. A and B are domains of lists, A is of the form a`*`, B is of the form b`*` and a is compatible to b.

8. A is the polymorphic domain of empty list and B is a domain of lists of any kind of elements.

9. A and B are domains of tuples, A has been defined as an extension of some another domain of tuple C, which is of the form `(`$c_1, \cdots, c_n$`)`, where c_i, for $1 \leq i \leq n$, denotes the domains of the C's components which include bound functions; B is of the form `(`$b_1, \cdots, b_n$`)`, where b_i, for $1 \leq i \leq n$, is the domain of the B's components, and each c_i is compatible to its corresponding b_i, for $0 \leq i \leq n$. The names of the tuple fields are not considered for the purpose of determining compatibility of domains.

10. A is a tuple domain of the form `(`A_1`)`, and A_1 is compatible to B.

11. A is domain expression, B is a union of domains of the form b_1 `|` $\cdots$ `|` b_n, and A is compatible to some b_i, for $1 \leq i \leq n$.

12. A and B are unions of domains, A is of the form a_1 `|` a_2 `|` $\cdots$ `|`a_n, B is of the form b_1 `|` b_2 `|` $\cdots$ `|`b_m, $n \leq m$, and a_j, for $1 \leq j \leq m$, is compatible to b_i, for $1 \leq i \leq n$.

13. A and B are domains of nodes, whose labels are identical.

14. A and B are domains of continuous functions, A is of the form a_1 `->` a_2, B is of the form b_1 `->` b_2, and b_1 is compatible to a_1, a_2 is compatible to b_2.

15. A is a constant domain of one quotation and B is the built-in domain `Q`.

16. A is the domain ? of undefined values.

From the above definitions of domains equivalence and compatibility follows that if domains A and B are equivalent then A is compatible to B, and vice-versa.

Chapter 4

Expressions

4.1 Introduction

SCRIPT expressions may be literal constants, variables, integer expressions, quotation expressions, logical expressions, list expressions, tuple expressions, node expressions, pattern expressions, inquiry expressions, conditional expressions, case-expressions, lam-abstractions, pattern-abstractions, let-expressions, functional applications or any well-formed combination of simpler expressions and operators.

Literal constants are members of the standard domains `N, Q, T` and `?`.

Variables are used to denote members of domains. Those which denote lists are usually, but not necessarily, suffixed by sequences of `*`'s or `+`'s. For instance, `x*` may denote a finite tuple of arbitrary size, whereas `x+` represents a non-empty finite tuple. Further information about lists represented by either `x*` or `x+` are provided by the associated domain declaration of `x`.

Integer expressions are built upon the operators `NUMBER, PLUS, MINUS, MULT, DIV, REM, NEG` and `SIZE`. Quotation expressions involve the operators `CAT`, `CONC` and `QUOTE`. Logical expressions use the operators `TRUTH, LT, LE, GT, GE, EQ, NE, NOT, AND` and `OR`. Tuple expressions are expressions whose results are tuples.

Every expression enclosed in parentheses is a tuple. However, parentheses may still be used to group terms of a formula, because all operators or functions that require a parameter of type, say `A`, accepts parameters of type `(A)` , which is automatically coerced to `A`. List expressions are expressions whose results are lists. The list operators are: `HEAD, TAIL, CAT, PRE, SIZE, EL` and `CONC`.

A node expression is of the form `[` $e_1 \cdots e_n$ `]`, where e_i, for $1 \leq i \leq n$, are expression of any type.

4.2 Inquiry Expressions

Expressions of the form e_1 `EQ` e_2 are used to test whether or not two expressions denote the same value. This expression evaluates `TT` if e_1 and e_2 have the same non-functional

value, FF if e_1 and e_2 are functional values or denote distinct values, and *bottom* if either e_1 or e_2, or both are *bottom*. The negation of e_1 `EQ` e_2 is written as e_1 `NE` e_2.

The operators `EQ` and `NE` only allow to test *values* denoted by expressions. However, in many cases it is desirable to investigate the structure of a value rather than the value itself. For example, sometimes it is important to ascertain whether the value denoted by an expression is a tuple or a node. To that end, $\mathcal{SCRIPT}$ makes available the inquiry operator `IS` and the associated pattern capabilities[10].

The operation e `IS` p performs a pattern-matching operation in order to check whether expression e has the particular "form" or structure described by pattern p. In essence, if the value denoted by e can be structured according to the rules dictated by p, then the result of the expression above is `TT`. Otherwise, it is `FF`. It also is `FF` if e is undefined. The result is *bottom*, if e denotes *bottom*.

4.3 Pattern Expressions

A pattern-expression can be the undefined value `?` (which matches any type of value), an identifier (which is generally treated as `?` in the operation `IS`), a literal constant (which matches itself) or a combination of simpler pattern-expressions and pattern construction operators. Notice that because pattern `?` matches anything to test for the undefined value itself, the operator `EQ` must be used instead of `IS`.

If $p, p_1, \cdots, p_n$ are pattern-expressions, then new patterns can be built up as follows:

1. `(`$p_1, \cdots, p_n$`)` - to match tuples with n components.
2. p`*` - to match lists with zero or more components.
3. p`+` - to match lists at least one component.
4. `< >` - to match empty lists.
5. p_1 `PRE` p_2`*` - to match lists with at least one component.
6. t_1 `CAT` t_2 - to match tuples that can be split into two tuples whose domains are those of t_1 and t_2, respectively.
7. `[`$p_1 \cdots p_n$`]` - to match nodes. Pattern $p_1, \cdots, p_n$ are required to be identifiers or literal constants from which the node label can be determined. A matching occurs if the value being tested is a node with the same label.
8. `NUMBER` p`+` - to match numbers.
9. `TRUTH` p`+` - to match truth-values.
10. `QUOTE` p`*` - to match quotations.

4.4 Conditional Expressions

Conditional expressions are of the form t`->` e_1`,`e_2, where t is an expression which evaluates to `TT`, `FF`, `?` or *bottom*, and e_1 and e_2 are arbitrary expressions. The expression above is equivalent to e_1 if t denotes `TT`; equivalent to e_2 if t stands for `FF`; equivalent to `?` if t evaluates to `?`, and equivalent to *bottom* if t represents *bottom*.

4.5 LAM-Abstractions

Anonymous non-recursive functions are specified by the operation `LAM` x`.`e. Usually, `x` is an identifier or a tuple of identifiers, and expression e is an arbitrary expression. The operator `LAM` binds x in the scope of the expression e, and denotes the function whose type is `A -> B`, if x`:A` and e`:B`.

Anonymous *recursive* functions are defined via the *fixed point* operation `Y(LAM` x `.` e`)`, where `Y` is the *Paradoxical Combinator* [5] defined as

```
Y = LAM (LAM b. a(b(b)))(LAM b. a(b(b))).
```

Expressions defining functions as above are called "LAM-abstractions".

4.6 Pattern Abstractions

Pattern-abstraction is a generalization of the notation for LAM-abstractions. The basic idea is to allow patterns to occur in binding contexts, such as in `LAM p.e`, where `p` is a pattern-expression.

The pattern-abstraction's binding mechanism provides a powerful mean of extracting components of a value. If, for example, e is a $\mathcal{SCRIPT}$-expression such that e `IS <x1,x2>` is equivalent to `TT`, then `x1` and `x2` will be bound to the first and second components of tuple e, respectively, in the scope of expression e_1 in `(LAM <x1,x2> .` e_1`)(`e`)`. If e `IS <x1,x2>` denotes `FF`, then the value of this function application is `?`. Notice that occurrences of `x1` and `x2` in the inquiry operation `IS` above are treated as `?`, but their occurrences in the pattern-abstraction are not. In fact, they get bound to the corresponding components of e.

Another illustration is the following pattern-abstraction application:

`(LAM [x1 x2] .` e_1 `)(` e_2 `)`

which binds `x1` and `x2` to the immediate subtrees of e_2 in the scope of e_1, if e_2 `IS [x1 x2]` is equivalent to `TT`.

In summary, the application of the pattern-abstraction `LAM` p`.`e, to argument a, where p is a pattern, e and a are expressions, produces the following result:

1. if the evaluation of e effectively needs the value of at least one of the identifiers occurring in p then:
   - if a `IS` p, the identifiers in p are bound to corresponding values in the structure of a, and then the body e is evaluated.
   - if a `IS` p produces `FF`, the result of the function application is `?`.
2. if no identifiers in p are needed to evaluate e, the pattern-matching is not performed.

If $p, p_1, \cdots, p_n$ are pattern-expressions, `x, x*` and `x+` are identifiers, and the pattern matching operation a `IS` p produces `TT`, then the bindings produced by the application of the pattern-abstraction `(LAM` p`.`e`)(`a`)` are recursively defined as follows:

1. if p is a literal constant or the empty list symbol (`<>`), no bindings result.

2. if p is identifier possibly decorated by `*`'s and `+`'s, then that identifier is bound to a.

3. if p is of the form `(`$p_1, \cdots, p_n$`)`, then the identifiers in p_i, for $1 \leq i \leq n$, are properly bound to the corresponding components of a.

4. if p is of the form p_1 `PRE` p_2, then the identifiers in p_1 are properly bound to the first element of the list a, and the identifiers in p_2 to the tail of a.

5. if p is of the form `[`$p_1 \cdots p_n$`]`, then the identifiers in p_i, for $1 \leq i \leq n$, are bound to the corresponding parts of node a.

6. if p is of the form `NUMBER x+`, then `x+` is bound to a list of quotations containing the decimal digits of number a.

7. if p is of the form `TRUTH x+`, then `x+` is bound to the list of quotations either `<"T","T">` or `<"F","F">` depending on the value of a.

8. if p is of the form `QUOTE x*`, then `x*` is bound to the list of quotations containing the characters of the quotation a.

A "recursive" pattern-abstraction is defined via `Y(LAM` p`.`e`)`, with the restriction that the value e `IS` p must be always "manifestly" `TT` [10].

In conjunction with LAM and IS, $\mathcal{SCRIPT}$ also provides the monadic operator `VAL` makes the enclosing `LAM` or `IS` *strict*, i.e., if the value of the pattern in front of a `VAL` is *bottom* then the enclosing LAM or IS also stands for *bottom*.

4.7 Case-Expressions

The `CASE` construct provides a mechanism to investigate the structure or "form" of a value, not the value, denoted by an expression according to a number of patterns and to produce as a result the expression which is associated with the first pattern that corresponds to the given value structure. The case-expression is of the form:

`CASE` e
  `/`p_1 `->` e_1
  ...
  `/`p_n `->` e_n
`END`

where $e, e_1, \cdots, e_n$ are ordinary expressions and $p_1, \cdots, p_n$ are patterns. The entire construct is equivalent to the following conditional expression:

e `IS` p_1 `->` `(LAM` p_1`.`e_1`)(`e`),`
  ...
e `IS` p_n `->` `(LAM` p_n`.`e_n`)(`e`), ?`

In essence, the structure of the value denoted by e is inquired in accordance with patterns $p_1, \cdots, p_n$. Then the first pattern, say p_i, that corresponds to the structure of e is

used to decompose e via a pattern-abstraction whose body is the corresponding expression e_i, and whose formal parameter is pattern p_i.

Note that the evaluation of e, the pattern-matchings and corresponding bindings are carried out before the evaluation of any of the case clauses. The case pattern-matching is strict in order to give a consistent rule on how case expressions are evaluated.

4.8 Let-Expressions

Large expressions can be broken into small pieces by means of let-expressions, which associate names with expressions and allow the use of these names in scope of a given expression.

Expressions of the form `LET` p `=` e' `IN` e, where p is an identifier, is the general form for defining non-functional values, but it can also define de-sugared functions. In this case, the definition p `=` e' may be deemed as a mechanism for giving a name to an expression e so that p can be used as an abbreviation for e' in the scope of the definition. When p is a more complicated pattern, the definition p `=` e' provides a mechanism to implicitly check the structure of e', and to decompose its value according to the structure depicted by p. For instance, if `x* IS (? PRE ?*)` is `TT` then the definition `x1 PRE x1* = x*` associates the name `x1` with the head of list `x*`, and `x1*` with the tail of `x*` throughout the definition's scope.

The general form of a let-expression is:

`LET` a_1 `=` e_1 `LET` a_2 `=` $e_2 \cdots$ `LET` a_i `=` $e_i \cdots$ `LET` a_n `=` e_n `IN` e

which defines a_i, for $1 \leq i \leq n$, whose scope are the expressions e_i and e. Each a_i mentioned above may occur in the form of *pattern binding* or *function definition.*

In case of *pattern binding*, each a_i must be a pattern expression. The value of expression e, which occurs in the corresponding `IN`-clause, is evaluated in the scope of the bindings of the identifiers in each pattern a_i to the corresponding structures of e_i.

Pattern bindings are used to decompose values into their components according to their structure. The expression

`LET` p_1 `=` e_1 `LET` p_2 `=` $e_2 \cdots$ `LET` p_i `=` $e_i \cdots$ `LET` p_n `=` e_n `IN` e

is equivalent to the following pattern-abstraction:

- Non-recursive let:

  `(LAM (`$p_1, \cdots, p_n$`) . `e`)(`$e_1, \cdots, e_n$`).`

- recursive let:

  `(LAM (`$p_1, \cdots, p_n$`) . `e`)( Y(LAM(`$p_1, \cdots, p_n$`) . (`$e_1, \cdots, e_n$`)))).`

The above equivalence relation requires that pattern-matching in let-expressions to be lazy. This means that the patterm conformance checking and the correspondig bindings of the identifiers in patterns p_i, for $1 \leq i \leq n$, are performed only if needed in the evaluation of expression e.

In case of *function definition*, each a_i is a function header of the form

f`(`p_1`:`d_1`)` $\cdots$ `(`p_n`:` d_n`)` `:` d

where f is the function head (usually an identifier), $p_1, \cdots, p_n$ are pattern-expressions, $d, d_1, \cdots, d_n$ are optional domain expressions. Function definitions are used to introduce a named pattern-abstraction whose formal parameters are patterns to be matched and bound to the corresponding parts of the actual parameters, and whose bodies are specified after the corresponding = sign. In other words, the expression

LET f(\p_1:d_1) $\cdots$ (p_n: d_n) : d = e

is tantamount to

LET f = LAM p_1 . LAM p_2 . $\cdots$ LAM p_n . e.

The function head f may be a function identifier, a term of the form D.g or g^, where D is a domain name and g if a function identifier, which is said to be bound to D. The symbol ^ has to do with polymorphism and dynamic binding of functions. It indicates a reference to the previous redefinition of a domain bound function.

Non-curried functions are specified as in

LET f(p_1:d_1, $\cdots$,p_n: d_n) : d = e.

Notice that because tuples of patterns are perfectly good patterns, one may write

LET <e'_1, $\cdots$,e'_n> = <e_1, $\cdots$,e_n> IN e

in place of a list of definitions of the form

LET e'_1 = e_1 $\cdots$ LET e'_n = e_n IN e.

A sequence of let-clauses of the form

LET f_1 = e_1 LET f_2 = e_2 $\cdots$ LET f_i = e_i $\cdots$ LET f_n = e_n IN e

where each f_i, for $1 \leq i \leq n$ is a function designator, may also be used to introduce collections of mutually recursive definitions. The syntax makes no distinction between recursive and non-recursive functions.

A let-expression of the above form is equivalent to

(LAM (f_1,$\cdots$,f_n).e)(e_1,$\cdots$,e_n)

if none of the f_i, for $1 \leq i \leq n$, is recursive, otherwise it is equivalent to:

(LAM (f_1,$\cdots$,f_n).e) (Y(LAM(f_1,$\cdots$,f_n).(e_1, $\cdots$,e_n))).

In a non-recursive definition

LET x = $_1$e in e_2

the domain of expression e_1 must be compatible to the domain of x. Whereas, in a recursive definition

LET x = $_1$e in e_2

the domain of expression e_1 must be equivalent to the domain of x.

In general, actual parameters must be compatible to their corresponding formal parameters.

4.9 Updating Functions

It is quite common in denotational semantic descriptions to define certain functions in a stepwise fashion. For instance, Initially, a function is defined to be undefined ("?") or another constant for all values of its argument. Then, the elements of this function are gradually "updated" for certain values of the argument as the definition progresses. Note

that the term "updating function" is just an abuse of notation. A new value is always produced as expected in the functional paradigm. Consider, for example, the domain `S` of stores commonly used in standard denotational semantic descriptions [7]. Stores are frequently modelled as:

`S = Loc -> Sv`,

where `Loc` is the domain of locations and `Sv` that of storable values. Initially, the store is assumed empty and a function `s:S` is defined as follows to reflect that fact:

`s loc = "unused"`,

where `"unused"` is a special value in the domain `Sv`. Later, when the value denoted by a given expression `e` is to be associated with a given location, say `a:Loc`, in the mapping `s`, function `s` is "updated" to `s{`a`=e}` such that for any `x: Loc`:

$$\texttt{s\{}a\texttt{=e\}(x)} = \begin{cases} e & \text{if } \texttt{x = a} \\ \texttt{s(x)} & \text{otherwise} \end{cases}$$

In $\mathcal{SCRIPT}$, an updating function consist of an expression, which must possess functional type, followed by a sequence of one or more "updatings" as in:

$\texttt{f}\{x_1 = e_1, \cdots, x_n = e_n\}\{\texttt{g}\}$

where $e_1, \cdots, e_n$ are arbitrary expressions in the domain `A`; $x_1, \cdots, x_n$ are expressions denoting members of `B`, and `f, g` have type `A -> B`. The meaning of the "updating" function above is given by:

`LAM x.` $\texttt{g(x) NE ? -> g(x), x EQ } x_1 \texttt{ -> } e_1\texttt{, } \cdots\texttt{, x EQ } x_n \texttt{ -> } e_n\texttt{,f(x)}$

4.10 Function Application

4.10.1 Sequential Expressions

Sequential expressions are combinations of $\mathcal{SCRIPT}$ expressions via the so-called sequential operators, which provide functional compositions in the way they are commonly used in denotational semantic descriptions.

The expression $f\ g\ e$, where g and f are expressions denoting functions and e is an arbitrary expression, is construed as $(f(g))(e)$.

It is often the case that the association of operators in the opposite way would be much more convenient. In particular, expressions of the form $f(g(e))$ are so common in denotational semantic descriptions that it is convenient to write them as $f;g;e$ in order to avoid excess of parentheses. The scope of the operator `;` extends until the end of the expression.

Traditional functional compositions are written as f `$` g so that f `$` g `(`e`)` is construed as $f(g(e))$.

4.10.2 Parameter Passing

$\mathcal{SCRIPT}$ is a pure, non-strict (or lazy) functional language so the evaluation of function arguments is delayed until they are needed. This means that the evaluation of the function's

body is initiated before the evaluation of the arguments. Note that in pattern-abstraction, the pattern-matching mechanism and the associated binding process are also delayed until the variables in the pattern are needed in the abstraction's body.

The domain of each actual parameter must be compatible to the domain of the corresponding formal parameter. Even in the case of tuple parameters, the binding mechanism always preserves the value of the actual parameter. So, if a tuple value is bound to a formal parameter whose domain is a base of that of the actual parameter, its whole value is delivered to the the function, although only its base part is accesible. Any further binding to the value passed still preserves the value originally passed. In the example below,

```
DOMAINS
  A = (x : N, y : N) ;
  B = A EXT (z : N) ;
DEFINITIONS
  DEF f(a) : A = a{x=3}
  DEF g(b) : B = ... LET a1 = f(b) ...
```

the definition `a1 = f(b)` binds the value of the 3-element tuple `b` to formal parameter `a`, which is a 2-element tuple. Function `f` returns a new 3-element tuple constructed by "updating" value bound to `a`. Then, the defintion `a1 = f(b)` binds `a1` to a 3-element tuple whose `x` component has value 3.

4.10.3 Domain Bound Functions

Functions can be associated with (or bound to) a given domain of tuples. For example, the function `push` below is defined to be bound to a domain named `Stk`.

```
DEF Stk.push(elem) : Stk =
    LET stk1 = elem PRE THIS IN  stk1
```

The application of `push` must always be associated with a variable (or object) in the domain `Stk` or in any extension of it. In the body of `push`, keyword `THIS` denotes the value of the currently associated object. The domain of the variable `THIS` is the domain of the object associated to each call.

A function that is bound to a domain of tuples, say `A`, may be redefined and rebound to any extension of `A`. In this case, the domains of the function in all of its redefinitions must be equivalent.

Function redefinitions form a hierarchical chain in accordance with the extension relation of the associated domains, and the valid redefinition is the lowest one in the hierarchy.

Within the body of a domain bound function `f`, the previous redefinition of the function in the hierarchy, that is, the definition bound by its direct base domain, can be referenced via the notation `f^`. In the example below, the function call `b.f(...)` computes the value of body of `B.f`, which uses `f^` to activate function `A.f`.

```
DOMAINS
   B = A EXT (N,N)

DEFINITIONS
   DEF A.f(x) = ...
   DEF B.f(x) = ... f^(x) ...
   DEF ...  b.f(x) ...
```

For other domains than that of tuples are *non-extensible*, their bound functions can never be redefined. The same holds for non-extended domains.

Except for *super call* notation `f^`, all domain bound functions references must be explicitly qualified by an object, including `THIS` object.

As a especial case, simple variables can be interpreted as zero-argument functions, and so can also be bound to domains.

4.10.4 Overloading of Functions

Function may have the identical names in the same scope, that is, names of functions can be overloaded. The declared domains of the arguments should be enough to resolve ambiguities.

The overloading resolution procedure that identifies the function being called in a functional application of the form `f` $a_1 \cdots a_n$, where a_i is in domain A_i, follows the steps:

1. from the function designator `f`, determine the set C of candidate functions. If `f` denotes a call to domain bound functions, only the functions bound to the indicated domain are considered; otherwise all functions with the same name in the scope should be entered in the candidate set;
2. for each actual parameter a_i, for $1 \leq i \leq n$, eliminate from C functions which have a corresponding formal parameter p_i in domain P_i such that A_i is not compatible to P_i. Note that only the first n formal parameters of `f` are considered in this process;
3. if cardinality of C is greater than 1, perform the following steps:
   (a) assign to each function in C the number of *nominal-matchings* occuring between the domains of the arguments and those of the corresponding parameters. A nominal-matching occurs when both domains are identical domain names.
   (b) keep in C only the functions to which the greatest nominal-matching number has been assigned.
4. if C is empty, the called function has not been declared in the calling scope. If cardinality of C is greater than 1, the function calling is ambigous.

For example, considered the following function definitions:

```
DOMAINS

   A = N ;
```

```
   B = Q ;
   C = N ;
   D = Q ;

DEFINITIONS

   DEF f a b = a    -- 1
   DEF f n b = n    -- 2
   DEF f q n = q    -- 3
```

Then:

- `f a1` refers to function 1;
- `f n1` refers to function 2;
- `f q1` refers to function 3;
- `f c1` is ambiguous; functions 1 and 2 are canditates;
- `f d1` refers to 3.

Chapter 5

The Module Structure

5.1 Introduction

A complete formal definition in $\mathcal{SCRIPT}$ consists of a main module along with zero or more (secondary) external modules which may either be compiled together with the main module or extracted from a library of already compiled modules. The main module is the one which contains the main function defined to model the meaning of an entire denotational semantic definition.

The basic function of a module is to allow related entities, such as domains and functions, to be grouped and then used by other modules. It provides a mechanism whereby details of certain domain and function definitions can be hidden from the user of the module, while making available a selected group of domains and functions for outside use.

There are three kinds of modules namely `PROJECT`, `SYNTAX` and `MODULE`.

5.2 `PROJECT` Module

The `PROJECT` module serve to define parameters and the environment in which the formal definition is to be evaluated.

This module may import only one function, which is undertood as the main function of the formal definition. In the example below, the main function is `elab-prog`, which is imported from module `Program`, defined to be the main function. The necessary domains can be imported freely.

For the sake of clarity, the signature of the main function may be redefined in the `DOMAINS` section of the `PROJECT` module provided that the new signature is equivalent to the old one. In the example below, `Prog-tree` and `Input-data` have been introduced only to emphasize to nature of the function arguments.

The `INFILES` and `OUTFILES` sections defines the association between files and the domains of argumentos of the the main function. This association is necessary to establish where the input arguments of the main function can be found, and where its result must

be recorded. The identification of each file is written within quotes ("), and it must obey the rules of the local operating system, where is $\mathcal{SCRIPT}$ language is implemented.

The COMPONENTS section defines the files that contain the various modules that comprise the formal definition.

```
PROJECT MiniL

IMPORTS
   Program(elab-prog, Prog, A)

DOMAINS
   elab-prog := Prog-tree -> Input-data -> A;
   Prog-tree  = (Prog)
   Input-data = (N) ;

INFILES
   Prog-tree = "name of the input file for (prog)"
   Input-data = "name for the input file for (n)"

OUTFILE
   A    = "name of the file in which the result of elab-prog"

COMPONENTS
    "Minil.lds", "Program.lds", "Env.lds", "Command.lds", "Expression.lds"
END Minil
```

5.3 SYNTAX Module

The SYNTAX module normally contains three sections:

- SYNTAX: defines the concrete and abstract syntaxes.
- LEXIS: defines the structure of lexical symbols.
- DOMAINS: specifies domains of non-terminal symbols.

The following example defines the grammar of a simple language:

```
SYNTAX MiniL
   prog  ::= "begin" dcls stmts "end" ;
   dcls  ::= dcl+ ;
   dcl   ::= "int" id+-"," ";" : [int id+] ;
   stmts ::= stmt* ;
   stmt  ::= id ":=" exp
           | "if" exp "then" stmts "fi"
```

```
           | "while" exp "do" stmts "end" ;
LEXIS
   UNIT    ::= id    : <"ID", id>
             | numeral : <"NM", numeral> ;
   id      ::= letter+ : QUOTE letter+
   numeral ::= digit+  : NUMBER digit+ ;
   letter  === "a" .. "z" ;
   digit   === "0" .. "9" ;

DOMAINS
    dcls, dcl   :  Dcl ;
    stmt, stmts :  Stmt ;
END MiniL
```

5.4 MODULE Module

Modules of type MODULE serve to encapsulate the definition of domains and functions and to establish the interface of comunication between modules. Each module is composed of three type of sections, namely:

- EXPORTS section: defines the entities exported by the module and their degree of encapsulation.
- IMPORTS section: lists the names of the entities imported to the modules along with their degree of visibility.
- DOMAINS section: declares domains, variables and functions.
- DEFINITIONS section: defines functions and other values.

A module is therefore a mechanism for explicit control of visibility. Every entity[1] declared in a module is private to the module, unless it is explicitly exported. The export list of a module defines its interface window with other modules in the semantic definition. Only domains and variables that are effectively defined in the module or imported to it from other modules can be exported. Conversely, the only identifiers visible in a module are those defined inside it or diretly imported to it.

Both exportation and importation of entities between modules can be *closed* or *open*, which are attibutes to specify the `visibility degree` of the exported information.

A closed exportation imposes on the exported entities the highest degree of information hiding with respect to their internal structures. A closed exportation may only correspond to closed importation. In this situation, when a domain identifier is imported, only its domain name and its extensibility property become available in the importing module.

[1] An entity is a variable, a domain or a function.

Thus the definition details of all closed exported domains remain encapsulated and hidden in the corresponding exporting module, but exported domains of tuple may be extended freely.

Thus, no further access, from the importing modules, is allowed to the domain definition.

When a variable is imported, only its name and *domains signature*, defined at the definition of the variable, are revealed at the importation point. The domain signature, or simply signature, of a variable is the domain expression directly associated with the variable, without unfolding the denotation of the domain occurring in this expression. All domains names mentioned in the signature of a variable must also have be explicitly imported if any reference to them is desired.

For example, consider the following definitions of modules M1 and M2:

```
MODULE M1
EXPORTS
   A, B, C, a, d, f, g

DOMAINS
   A =  B -> C ;
   B =  <Q,N> ;
   C =  N ;
   D =  B -> C ;
   d := <A,C> ;
   F  =  <Q,N> ;

DEFINITIONS
   DEF  a = LAM b. ...      -- signature of a is A
   DEF  d = ...             -- signature of d is <A,C>
   DEF  f(b) : C = ...      -- signature of f is B -> C
   DEF  g(b:F) : C = ...    -- signature of g is F -> C

END M1

MODULE M2
IMPORTS
   M1(B, C, a, d, f, g)
   ....
END M2
```

In module `M2` the domain names `B, C` and identifiers `a, d, f` and `g` are available and no knowledge regarding the definitions of `B` and `C` is allowed. The visible signatures of the imported variables are:

```
a : A
```

```
d : <A,C>
f : B -> C
g : F -> C
```

Although imported variables `a, d` and `g` cannot be effectivelly used anywhere within `M2`, because domains `A` and `F` were not imported along. On the other hand, function `f` can be used freely, but the definitions of imported `B` and `C` is not locally available. Note that in the scope of M1, the domains of `f` and `g` are structurally equivalent, but they cannot be considered so in the the scope of `M2`.

On the other hand, an open exportation gives the importing modules the oportunity to decide the degree of information hiding which is more convenient to its aplication. An open exportation does not reveal the entire structure of the exported entities; for example, the open exportation of a domain `B` autorizes access only to the first level of the domain denotation of `A`. The internal structure of any domain occurring in this denotation is remain completely encapsulated in the exporting modules, unless their open exportation and importation are explicitly specified too.

Open exportation or importation is indicated para the "`*`" sign in front of the entity name. In the example below, `A` is exported closed, `B` is exported open, `C` is imported closed and `D` is imported open from module `M1`.

```
MODULE M
EXPORTS A, *B ;
IMPORTS M1(C, *D)
  ...
END M
```

Closed and open re-exportation of entities are fres allowed, but the the visibility degree of these entities can only be decreased, never increased.

5.5 Example

Now the denotation definition of the toy language `MiniL` whose `PROJECT` and `SYNTAX` modules have been defined above is presented.

```
MODULE Program
EXPORTS elab-prog;
IMPORTS
   Env(State, A, *Ec, *Cc, state0, V, is-N, to-A)
   Command(Cmd, elab-cmds)
   Expression(Exp, elab-exp, Id, elab-id)
DOMAINS
   Prog = ["program" Body "end"] ;
   Body = ["read" Id Cmd+ "write" Exp];
```

```
   k : Ec; c : Cc;
DEFINITIONS
 elab-prog (prog) (i:N) : A =
   LET ["program" body "end"] = prog
   LET ["read" id cmd+ "write" exp] = body
   LET k = LAM v . is-N(v) ->
                   to-A(v), to-A("Error")
   LET c = LAM s. elab-exp(exp) k s
   LET s = state0.set(i,elab-id(id))
   IN  elab-cmds(cmd+)(c)(s)
END Program
```

Note that the continuation domains `EC` and `Cc`, which are defined in module `Env` below, are exported and imported opened in order to make the details of their internal structure known inside module `Program` so the continuation functions `k` and `c` above can be defined locally as it is usual in standard denotational semantics. If opened exportation were not allowed, it would be very inconvenient to work with continuation semantics. The module `Env` encapsulates everything concerned to the environment in which the semantics is modelled.

```
MODULE Env
EXPORTS
    State, state0, State.get, State.set, A,
    to-A, *Cc, *Ec, V, inc, is-N, is-undef;
DOMAINS
    State = (s:Id -> V);
    V     = [N] | ["undefined"] ;
    Cc    = State -> A;
    Ec    = V -> A ;
    A     = N | "Error" ;
DEFINITIONS
   DEF is-N(v)      : T  = v IS [n]
   DEF is-undef (v) : T  = v IS ["undefined"]
   DEF inc([n])     : V  = [ n PLUS 1 ]
   DEF to-A "Error" : A  = "Error"
   DEF to-A v       : A  =  LET [n] = v IN n
   DEF state0(id:Q) : V  = ["undefined"]
   DEF State.get(id:Q): V  = this.s(id)
   DEF State.set(n,id:Q) : State = this.s{id=n}
END env
```

The semantics of `MiniL` commands is defined as:

```
MODULE Command
```

```
EXPORTS
  Cmd, elab-cmds;
IMPORTS
  Expression(Exp, elab-exp, Id, elab-id)
  Env(*Ec, *Cc, V, State, state0, A,
      is-N, to-A, to-N)
DOMAINS
  Cmd = [Id ":=" Exp] | [Cmd+] | ["while" Exp "do" Cmd+] ;

DOMAINS
  c : Cc; k : Ec; s : State;

DEFINITIONS
DEF elab-cmds(cmd*)(c)(s) : A =
 CASE cmd*
   /<> -> s
   /cmd1 PRE cmd1* ->
     LET c1 = LAM s. elab-cmds(cmd1*)(c)(s)
     IN  elab-cmd(cmd1)(c1)(s)
 END

DEF elab-cmd(cmd)(c) (s) : A =
    CASE cmd
     /[id ":=" exp] ->
          LET q = elab-id(id)
          LET k = LAM v. is-N(v) ->
                    c (s{q=v}), to-A("Error")
          IN elab-exp(exp)(k)(s)

     /[cmd+] -> elab-cmds(cmd+)(c)(s)

     /["while" exp "do" cmd+] ->
        LET c1 = LAM s1 . elab(cmd)(c)(s1)
        LET k = LAM v .
                is-N(v) -> to-N(v) EQ 0 ->
                c(s), elab-cmds(cmd+)(c1)(s),
                to-A("Error")
        IN  elab-exp(exp)(k)(s)

      END
END Command
```

The semantics of MiniL expressions is defined as:

```
MODULE Expression

EXPORTS
  Exp, Id, elab-exp, elab-id;

IMPORTS
  Env(*Ec, V, State, state0, A, is-N, to-A, to-N)

DOMAINS
  Exp = ["0"] | [Id] | ["suc" Exp];
  Id  = ("ID", q) ;
  k   : Ec;
  s   : State;

DEFINITIONS

DEF elab-id(id) : Q =
    LET ("ID", q) = id IN q

DEF elab-exp(exp)(k)(s) : A =
    CASE exp
     /["0"] -> k(to-V(0))

     /[id] ->
       LET q = elab-id(id)
       LET v = s.get(q)
       IN  is-undef(v) -> to-A("Error"), k(v)

    /["suc" exp] ->
       LET k1 = LAM v . is-N(v) -> k(inc(v)),
                        to-A("Error")
       IN elab-exp(exp)(k1)(s)
    END
END Expression
```

5.6 Name Conflict Resolution

Name conflicts may arise when a module imports from two or more other modules. These conflicts are resolved by means of the `RENAMES` clause, which provides a local name for an imported entity. The example below shows the use of the clause:

```
MODULE A
  export x;
```

```
  ...
END A

MODULE B
  export x;
  ...
END A

MODULE C
  import A(x);
  import B(y renames);
  -- x is the x from A
  -- y is the x from B
  ...
END A
```

5.7 Type Discipline

1. Inquiring Expression: *Type*$[\![e IS\ p]\!]$ = `T`

   CONDICOES: ddd

   RULE: domain(p) must be compatible with domain(e)

2.

Chapter 6

Object-Oriented Programming in *SCRIPT*

6.1 Introduction

SCRIPT is considered an object-oriented language for denotational specification because it features the basic concepts of classes, objects, type inheritance, data encapsulation, information hiding, polymorphism, virtual functions and dynamic binding.

6.2 Class and Objects

The *SCRIPT* analogous to the classes of imperative object-oriented programming languages is the tuples. The module mechanism permits the encapsulation of domain definition and means to provide a well specified interface. Since domain of tuples implements classes, tuples as objects.

6.3 Inheritance

The device to extend tuples is in fact a mechanism to achieve inheritance. Any extended tuple is a descendent of its base tuple. Analogous to class inheritance, tuple extension is a mean to create a hierarchy of tuple types. An extended tuple domain heirs the structure of its base tuple along with all functions bound to the base domain.

6.4 Polymorphism

The allowed forms of polymorphism are *overloading* and *inclusion*.

Polymorphism of overloading occurs when more than one function receives the same name in the same scope. The types of the arguments must permit unambigous resolution of the name overloading. A set of functions with equal name may be viewed as a polymorphic

function that can be applied to arguments of number of different types, according to each function definition.

Polymorphism of inclusion is connected to the domain extension capability of tuples. Every function that has an argument of a tuple type also accepts arguments in domains which are extensions of that type. The extension relation between two domains of tuples is an **is-a** relation. This means that a tuple in an extended domain is always an instance of tuple in the corresponding base domain.

6.5 Data Encapsulation

Module is the mechanism to encapsulate data and exercise information hiding. The export/import facility of the language allows tight control of visibility and information hiding as required to achieve high degree of modularization.

6.6 Virtual Functions

Functions can be associated with or bound to domains. These functions are equivalent to the virtual functions of object-oriented imperative programming languages.

If the domain to which the function is bound is a tuple, then every application of this function must be qualified by an object whose domain is the bound domain or in an extension of it.

A domain bound function may be redefined in any extension of the associated domain. The redefined function is avaliable from that point it was defined downward in the domain hierarchy. By definition, if a function is bound to a tuple domain, then it is said to be bound to all its descendent, unless it is redefined.

6.7 Dynamic Binding

The conjunction of the concepts of polymorphism and virtual functions gives rise to *dynamic binding* of functions. Whenever the object that qualifies the call of a domain bound function is a formal parameter of type tuple, the function to be activated is the one currently bound to the domain of the current value of the actual parameter, whose type can be any extension of the domain of the formal parameter.

6.8 A Polymorphic Stack

Lists serve to model the concept of a stack whose operations are `push`, `pop`, `top` and `empty`. The following module, named `Stack`, defines the encapsulated domain `Stk`, the stack, the type of the stack element, and associated stack operations. Note that the elements of the stack have type `Stk-elem` which is an empty tuple. A client module of `Stack` may freely

extend `Stk-elem` to incorporate the type of information that actualy must go to the stack. Functions `push` and `pop` should handle the polymorphic elements of the stack properly.

```
MODULE Stack
EXPORTS
    Stk, stk0, Stk-elem, stk-elem0, Stk.push, Stk.pop, Stk.top, Stk.empty

DOMAINS
    Stk = (s:Stk-elem*)
    Stk-elem  = ()

DEFINITIONS
    DEF stk0 = (<>)

    DEF stk-elem0 = ()

    DEF Stk.push(stk-elem) : Stk = (stk-elem PRE THIS.s)

    DEF Stk.top : Stk-elem =
        THIS.empty -> ?, LET (stk-elem PRE stk1) = THIS IN  stk-elem

    DEF Stk.pop  : Stk = LET (stk-elem PRE stk1) = THIS IN  stk1

    DEF Stk.empty : T = (SIZE THIS.s) EQ 0

END Stack
```

Note that because the stack may have elements of differents types in the domain hierarchy that start with `Stk-elem`, it is wise to bind the value returned by function `pop` to a variable in the highest domain in the hierarchy and then explicitly test its form via a case construct.

```
MODULE User
IMPORTS
    Stack(Stk, stk0, Stk-elem, stk-elem0)

DOMAINS

    A-elem   = Stk-elem EXT (elem : N);
    B-elem   = Stk-elem EXT (elem1 : Q, elem2: T)
    X = (N | (Q,T))

DEFINITIONS
```

```
DEF test : X  =
    LET a-elem1 = stk-elem0 CAT (1)
    LET stk1    = stk0.push(a-elem1)
    LET b-elem1 = stk-elem0 CAT ("A", TT)
    LET stk2    = stk1.push(b-elem1)

    ..................

    LET a-elem = stk2.top
    LET stk3   = stk2.pop
    IN  CASE a-elem
          /stk-elem CAT (elem) -> elem
          /stk-elem CAT (elem1, elem2) -> (elem1,elem2)
        END

END User
```

Bibliography

[1] Lennart Augustsson and Thomas Johnsson. Lazy ML user's manual. Technical report, Chalmers University, Goteborg, Sweden, 1992. Department of Computer Science.

[2] D.M. Berry and R. L. Schwartz. Type equivalence in strongly typed languages: One more look. *ACM SIGPLAN NOTICES*, 14(9), 1979.

[3] Roberto S. Bigonha. *A Denotational Semantics Implementation System*. PhD thesis, University of California, Los Angeles, 1981. 428 pages.

[4] Richard Bird and Philip Wadler. *Introduction to Funcional Programming*. Prentice Hall International Series in Computer Science, 1988.

[5] W.H. Burge. *Recursive Programming Techniques*. ?, Reading, Mass, 1975.

[6] Luca Cardelli and Peter Wegner. On undertanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] M.J.C. Gordon. *The Denotational Description of Programming Languages - An Introduction*. Springer-Verlag, New York - Heiberg - Berlin, 1979.

[8] S.L.P. Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science, Englewood Cliffs, 1987.

[9] P. D. Mosses. *Mathematical Semantics and Compiler Generation*. PhD thesis, Oxford University Computing Lab, 1975. Programming Research Group.

[10] P. D. Mosses. SIS - A compiler-generator system using denotational semantics. Technical report, University of Aarhus, Denmark, 1978.

[11] Peter D. Mosses. Denotational semantics. In *Lectures Notes of the State of the Art Seminar on Formal Description of Programming Concepts – IFIP TC2 WG 2.2*, Rio de Janeiro, Brazil, April 1989.

[12] Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.

[13] D. Scott. Outline of a mathematical theory of computation. In *Proceedings of the 4th Princeton Conference on Information Sciences and Systems*, 1970.

[14] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, 1971.

[15] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Tech. mon. prg-6, Oxford University Computing Lab, Polytechnic Institute of Brooklyn, 1971.

[16] J.E. Stoy. Foundations of mathematical semantics. *Lecture Notes in Computer Science, Springer-Verlag*, 1979.

[17] R.D. Tennent. A denotational definition of the programming language PASCAL. Technical memo, Oxford University Computing Lab, 1978. Programming Research Group.

[18] David Turner. An overview of miranda. *ACM SIGPLAN NOTICES*, 21(12):158–166, 1986.

[19] David A. Watt. *Programming Languages Syntax and Semantics*. Prentice Hall International Series in Computer Science, New York, 1991.

[20] G. Winskel. *Semantics of Programming Languages*. MIT Press, 1993.