LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ID

Language Reference Manual

Version 90.1

Rishiyur S. Nikhil

Computation Structures Group Memo 284-2

July 15, 1991

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Id
Language Reference Manual
(Version 90.1)

Rishiyur S. Nikhil†

July 15, 1991

Computation Structures Group
Laboratory for Computer Science
Massachusetts Institute of Technology

545 Technology Square,
Cambridge, MA 02139, USA

Abstract

Id is a general-purpose parallel programming language designed by members of the Computation Structures Group in MIT's Laboratory for Computer Science, and is used for programming dataflow and other parallel machines.

The major subset of Id (syntactically distinguishable) is a pure functional language with non-strict semantics. Features include: higher-order functions, a Milner-style statically type-checked polymorphic type system with overloading, user defined types and pattern-matching notation, lists and list comprehensions, arrays and array comprehensions, and facilities for delayed evaluation.

The non-functional aspects of Id include I-structures and M-structures (for both arrays and user-defined types), and input/output. With respect to the functional subset, programs with I-structures remain deterministic but may not be referentially transparent, and programs with M-structures and i/o may even be non-deterministic.

Id programs are implicitly parallel to a very fine grain. Some programs with M-structures and i/o may need explicity sequencing, for which facilities are provided.

---

† Inquiries about Id may be directed by mail to Prof. Arvind at the above address, by email to `id@lcs.mit.edu`, or to the author. Author's current address: Digital Equipment Corporation, Cambridge Research Laboratory, One Kendall Square, Bldg 700, Cambridge MA 02139, USA; Email: `nikhil@crl.dec.com`; Telephone: (617) 621 6639. This work was done while the author was at MIT.

Contents

1 Introduction

Id is a parallel programming language designed by members of the Computation Structures Group of MIT/LCS. It is used for programming dataflow and other parallel machines.

Id is a language with three layers. The major subset of Id is a purely functional language; this subset is described first, in Section 2. The second layer extends this with I-structures which are described in Section 4. The third layer extends this with M-structures which are described in Section 5. Some general aspects of these non-functional extensions are described in Section 3.

Id traces its roots back to 1978 [1]. Since then, versions of Id have run on simulated dataflow machines, and more recently on real dataflow hardware and Unix workstations. Id/83s [4] was a first cut at a major redesign of the language, based on contemporary ideas in functional languages. It briefly acquired the name Id Nouveau (1986) [6, 2], and then reverted to Id in 1988 [3].

Id continues to be a research language. Current investigations include better constructs to express non-deterministic computations, I/O, resource-management *etc.*

This document is not a tutorial on Id. For a tutorial introduction, the reader is referred to [5].

Appendix C lists the incompatibilities between this version of Id and previous versions.

Acknowledgements

Many people—past and current members of the Computation Structures Group, and colleagues elsewhere—have participated in the design of Id. Major contributors include: Shail Aditya, Arvind, Paul Barth, David Culler, Kattamuri Ekanadham, Steve Heller, James Hicks, Vinod Kathail, Rishiyur Nikhil, Keshav Pingali, Ken Traub, and Jonathan Young.

Id is also heavily influenced by various functional languages in the ISWIM-ML-SASL family.

2 Functional Id

This section describes the purely functional (and referentially transparent) subset of Id.

2.1 Expressions, Statements and Types

Two major syntactic categories in Id are *expressions* and *statements*.

Every expression denotes a *value*. In this manual, we use the generic symbols "`e`", "`e1`", *etc.* to designate arbitrary expressions.

Statements appear in the top-level of programs, in blocks, *etc.* Statements are usually identifier bindings and declarations of new types.

Id has a polymorphic type system. Every expression and statement must "type-check", *i.e.,* satisfy certain type-rules; these are explained as each construct is introduced. Types are described by *type-expressions*. In this manual, we use the generic symbols "`t`", "`t0`", *etc.* to refer to types.

Type-checking in Id is done by *type inference*, *i.e.,* in general the programmer is not required to declare the types of identifiers or expressions—the type-checker automatically deduces them from the context. However, for readability, for better error-messages and to assist in overloading resolution, there is a facility for declaring types of identifiers (using `typeof` statements, see Section 2.10).

For explaining the type rules in this manual, we use the notation:

```
e  :: t
```

which is pronounced "`e` has type `t`", *i.e.,* the value of expression "`e`" lies in the set of values denoted by type "`t`". This notation is only a device for this manual; it is not part of the language.

2.2 Programs

A program is a collection of statements:

```
STATEMENT ;
...
STATEMENT ;
```

The statements collectively define an environment in which top-level expressions may be evaluated. The statements, together with a top-level expression, have the semantics of a block (see Section 2.25).

2.3 Parentheses and Grouping

Any expression or type-expression may be enclosed in parentheses. This may be done to override precedence, or merely for visual clarity.

```
( 2 + 3 ) * ( 4 - (f x))
```

```
(btree (btree N))
```

Parentheses are also used for "quoting" binary infix operators (see Section 2.13), for denoting the `void` value (see Section 2.15), and for grouping statements (see Section 3.7.)

2.4 Semicolons

Semicolons may be used either as a separator between statements, or as a terminator at the end of a statement.

2.5 Comments

Comments begin with "%" and can contain any text up to the end-of-line:

```
% anything goes till the end of the line
```

We recommend the guidelines on page 348 of the Common Lisp manual (Guy L. Steele, Jr., Digital Press, 1984) for commenting code, except that Id has "%" instead of Lisp's ";" as the comment character.

2.6 Identifiers

Identifiers may contain alphabetics, digits, underscores (_), question marks (?), single quotes (') and tildes (~) in any order. Examples:

```
x
x'
harry
desmond_2_2
2D_array
nil?
done?
```

The lexical syntax for identifiers overlaps with the lexical syntax for reserved words (Sections 2.6.1), numbers (Sections 2.17 and 2.18), character constants (Section 2.19) and the special underscore token "_" (Sections 2.27, 4.3.3 and 5.3.3). A lexical token is read as an identifier only if it is not in one of these categories.

Upper- and lower-case letters are equivalent in identifiers.

2.6.1 Reserved Words

The following words are reserved and may not be used as identifiers:

`abstype`	`for`	`rep`
`accumulate`	`fun`	`seq`
`and`	`if`	`sequential`
`array`	`in`	`then`
`bound`	`instance`	`to`
`by`	`instances`	`type`
`case`	`matrix`	`typeof`
`def`	`M_array`	`typesyn`
`defsubst`	`M_matrix`	`unbounded`
`do`	`M_vector`	`unless`
`downto`	`next`	`upfrom`
`downfrom`	`of`	`vector`
`else`	`or`	`when`
`finally`	`record`	`while`

In addition, the following families of words are reserved:

k`_`n`D_arrays`	k`_`n`D_M_arrays`
k`_vectors`	k`_M_vectors`
k`_arrays`	k`_M_arrays`
k`_matrices`	k`_M_matrices`
n`D_array`	n`D_M_array`

for each $k \geq 1$ and $n \geq 1$.

Upper- and lower-case letters are equivalent in reserved words.

2.6.2 Standard Identifiers

Standard identifiers are not reserved words—they can be redefined by the programmer, and the set of standard identifiers will continue to grow as more and more useful library functions are identified and implemented. To enhance readability and reusability of code, the programmer is strongly advised not to redefine them. See Appendix A for a listing of standard identifiers.

2.7 Types

In this manual, we use the generic symbols "`t`", "`t1`", *etc.* to designate arbitrary type expressions.

Types are denoted by *type-expressions*, which are either *Type Variables*:

`*3  *0  *13`

or N-ary *Constructed Types* ($N \geq 0$):

`type-constructor t1 ... tN`

A type constructor is an identifier (*e.g.,* `bool`, `list`, `tree`) or one of the array reserved words (*e.g.,* `array`, `vector`, `matrix`, `nD_array`, `nD_M_array`).

The identifiers used for type constructors may overlap with identifiers used for values. Since type expressions occur only in specific contexts, there is no ambiguity. For example, there is a type called `float`, and there is also a standard identifier `float` representing the function that converts integers into floating-point numbers.

Some pre-defined 0-ary constructed types (also called *Type Constants*):

`void`		
`char`	or	`C`
`bool`	or	`B`
`int`	or	`I`
`float`	or	`F`
`string`	or	`S`
`symbol`	or	`SYM`

The types in each row are synonyms.

Some pre-defined constructed types:

- **Array Types:**

`1D_array t`	`vector t`	`array t`
`2D_array t`	`matrix t`	
`3D_array t`		
...		

  The types in each row are synonyms.

- **List Types:**

  `list t`

Certain pre-defined constructed types also have special syntax:

- **Tuple Types:**

  `t0 , ..., tN`

- **Function Types:**

  `t0 -> t1`

  The "`->`" type operator associates to the right, so that the parentheses can be omitted in the following type-expression:

  `int -> (int -> bool)`

Additional pre-defined constructed types are listed in Sections 4, 5, 6 and A.13.

2.7.1 Precedence in Type Expressions

Type application binds tighter than "`->`", which binds tighter than comma. In each of the following examples, the parentheses may be dropped:

```
(btree int) -> int
(list int),int
(int -> int),int
```

2.7.2 Polymorphic Types

A type containing a type variable is a *polymorphic* type, *e.g.,* the type of "`:`", the list constructor, is:

`*0 -> (list *0) -> (list *0)`

The type variable stands for "any type", indicating that "`:`" can construct lists of any type. However, all occurrences of a type variable in a polymorphic type must be instantiated *uniformly*. For example, all these are valid instantiations of the type of the list constructor:

`int -> (list int) -> (list int)`

for building lists of integers, or:

`bool -> (list bool) -> (list bool)`

for building lists of booleans, or:

```
(int->bool)          ->
(list (int->bool)) -> (list (int->bool))
```

for building lists of integer-to-boolean functions. However, the following is not a valid instantiation:

`char -> (list bool) -> (list int)`

since the type variable `*0` is instantiated non-uniformly to `char`, `bool` and `int`, respectively.

While this may seem restrictive compared to, say, Lisp, in fact it is not, because disjoint union types (Section 2.11) give a way of packaging different types into a common type in a type-safe manner.

2.8 Overloading

There are many operators and identifiers that are not polymorphic, but overloaded. A polymorphic function is a single function that works on an infinity of types (each possible instantiation of its type variables). An occurrence of an overloaded identifier, on the other hand, is a syntactic shorthand for one of a small set of identifiers that have different types; which one it actually stands for depends on the context in which it is used—this resolution is performed automatically by type checking.

For example, the operator symbol "+" represents one of the following functions:

```
plus~int    :: int -> int -> int
plus~float  :: float -> float -> float
```

The identifier `length` represents one of the following functions

```
length~string :: string -> int
length~list   :: (list *0) -> int
```

For each use of an overloaded operator or identifier, the type-checker will attempt to infer the particular type at which it is used from the surrounding context, so that the particular function it represents is known. If it is unable to do so, an error is flagged. In this situation, the programmer must assist the type checker either by replacing the overloaded identifier by the intended non-overloaded one, or by using an explicit type declaration (Section 2.10) to make the type unambiguous.

We use the following syntactic convention to relate an overloaded identifier to the corresponding non-overloaded identifiers. An overloaded identifier `foo` represents one of the identifiers:

```
foo~type1
foo~type2
...
```

Appendix B lists many overloaded operators and identifiers, and their corresponding non-overloaded identifiers.

User-defined overloading

The user may declare that an identifier `x` is overloaded by using the statement:

```
overload x = t ;
```

The type of each non-overloaded instance of `x` is only allowed to be an instance of the type `t`. For example:

```
overload plus = *0 -> *0 -> *0 ;
```

Non-overloaded instances of an identifier `x` may be declared using the statement:

```
instance x = x~a, x~b, ... ;
```

The reserved word `instances` is a synonym for `instance`. Here, `x` must already be known as an overloaded identifier (*i.e.,* previously declared using an `overload x = t` declaration). The types of `x~a`, `x~b`, ... must be known, and must each be an instance of the overloaded type `t`.

There may be multiple `instance` declarations for the same overloaded identifier. Example:

```
instances plus = plus~int, plus~float;
```

or

```
instance plus = plus~int ;
...
instance plus = plus~float ;
```

All instances of an overloaded identifier should have mutually exclusive types, *i.e.,* for each pair of instances `x~a` and `x~b`, their types should not be unifiable.

Polymorphism of overloaded identifiers

Some overloaded identifiers represent polymorphic functions. For example, `length` represents:

```
length~string :: string    -> int
length~list   :: (list *0) -> int
```

and the latter function is polymorphic. However, a particular occurrence of `length` cannot be used polymorphically. Thus, the following program will not type-check:

```
{ f = length
In
  (f integer_list),(f bool_list) }
```

even though it would type-check if we use `length~list` instead. Two *occurrences* of `length` can, of course, be used at different types.

2.9 Type synonyms: typesyn

A new type constructor may be declared as a synonym for an existing type. The statement:

```
typesyn tx tv1 ... tvN = t ;
```

declares `tx` to be a new type. The `tvJ`'s are optional type variables ($N \geq 0$). Examples:

```
typesyn S           = string ;
typesyn complex     = (float,float) ;
typesyn code_mem    = array (op,src,dests) ;
typesyn eq_func *0 = *0 -> *0 -> bool ;
```

The type variables in the right-hand side, if any, must be a subset of the type variables mentioned in the left-hand side.

2.10 Type Declarations: `typeof`

An identifier's type may be declared anywhere in its scope. The statement:

```
typeof x = t ;
```

asserts that identifier `x` denotes a value of type `t`. Example:

```
typeof map~list = (*0 -> *1) ->
                  (list *0)  -> (list *1);
```

Since Id's type-checker automatically infers the types of all identifiers, user-specified type declarations are not generally necessary. However, we strongly recommend their plentiful use because:

- They make programs more readable;
- Error messages from the type-checker will be more localized, and hence more helpful.

Type declarations are sometimes necessary to assist the type-checker in resolving overloaded identifiers.

Note: a type declaration statement does not *introduce* any new identifiers or types.

2.11 Algebraic Types

Algebraic types are also called "disjoint union" types. New algebraic types are defined by the statement:

```
type tx tv1 ... tvN = disj1 | ... | disjM;
```

Here, `tx` is the name for the new type. Its optional N (≥ 0) type parameters are specified by the type-variables `tvJ`. Its M (≥ 1) disjuncts are specified by the `disjJ`s, each of which has the form:

```
tcons t1 ... tL
```

Here, `tcons` is an identifier and represents a new L-adic (≥ 0) *Constructor*. Each `tJ` is a type-expression constraining the type of the J'th argument of the constructor. Thus,

```
tcons :: t1 -> ... -> tL -> (tx tv1 ... tvN)
```

The type variables in the right-hand side (all disjuncts), if any, must be a subset of the type variables mentioned in the left-hand side.

Implicitly defined with each L-adic constructor are also L *field selectors* `tcons_1`, ..., `tcons_L`. Assuming we have an expression `e` that evaluates to

```
v    :: tx tv1 ... tvN
```

then the "dot-notation" expression:

```
e.tcons_J
```

checks that `v` is indeed of the form:

```
tcons v1 ... vL
```

i.e., checks that it is in the expected disjunct, and then returns `vJ`. If `v` is a different disjunct, a run-time error occurs, producing the error value $\perp$ (see Section 2.38).

Implicitly defined with each constructor `tcons` is also a predicate function:

```
tcons?   ::  (tx tv1 ... tvN) -> bool
```

that tests whether a value of that type is in that disjunct.

Examples

Lists of integers:

```
type ilist = Inil | Icons int ilist;
```

Implicitly defined constructors:

```
Inil    :: ilist
Icons   :: int -> ilist -> ilist
```

Implicitly defined field selectors:

```
ICons_1    :: ilist . int
ICons_2    :: ilist . ilist
```

both of which produce the error value $\perp$ if `e` evaluates to `INil` (see Section 2.38).

Implicitly definied predicates:

```
Inil?   :: ilist -> bool
Icons?  :: ilist -> bool
```

Polymorphic lists:

```
type list *0 = Nil | Cons *0 (list *0);
```

Implicitly defined constructors:

```
Nil    :: (list *0)
Cons   :: *0 -> (list *0) -> (list *0)
```

Implicitly defined field selectors:

```
Cons_1    :: (list *0) . *0
Cons_2    :: (list *0) . (list *0)
```

both of which produce the error value ⊥ if `e` evaluates to `Nil` (see Section 2.38).

Implicitly defined predicates:

```
Nil?   :: (list *0) -> bool
Cons?  :: (list *0) -> bool
```

Polymorphic binary trees:

```
type btree *0 = Empty_btree
              | Bnode *0
                      (btree *0)
                      (btree *0) ;
```

Implicitly defined constructors:

```
Empty_btree :: (btree *0)
Bnode       :: *0 ->
               (btree *0) ->
               (btree *0) -> (btree *0)
```

Implicitly defined field selectors:

```
Bnode_1    :: (btree *0) . *0
Bnode_2    :: (btree *0) . (btree *0)
Bnode_3    :: (btree *0) . (btree *0)
```

all of which produce the error value ⊥ if `e` evaluates to `Empty_tree` (see Section 2.38).

Implicitly defined predicates:

```
Empty_btree?   :: (btree *0) -> bool
Bnode?         :: (btree *0) -> bool
```

2.12 Function Applications

Every function has type "`t0 -> t1`" for some argument type "`t0`" and result type "`t1`".

Assuming:

```
ef :: (t0 -> t1)
ex :: t0
```

then the application expression:

```
ef ex :: t1
```

denotes the application of a function (the value of `ef`) to an argument (the value of `ex`).

Application associates to the left. Thus, the following two expressions are equivalent:

```
e1 e2 e3 ... eN
```

```
(((e1 e2) e3) ... eN)
```

2.13 Prefix and Infix Operators

Some functions are designated by special symbols called *operators*. Unary prefix operator expressions are written:

```
op e
```

Binary infix operator expressions are written:

```
e1 op e2
```

All binary operators can be treated as ordinary identifiers by enclosing them in parentheses, *e.g.,*

```
(+) e1 e2
foldr_list (+) 0 list_of_N
```

This is one of the two special uses of parentheses in Id, where they are not used for grouping (the other is the notation for the `void` value, Section 2.15).

The unary prefix operator "`-`" cannot be similarly treated as an ordinary identifier by enclosing it in parentheses, because "`(-)`" stands for the value of the binary version. For example, if the programmer needs a unary integer minus function, this can be obtained by partially applying the binary function to the constant "`0`":

```
((-) 0)
```

When applied to some `x`, this becomes "`0-x`", which is equivalent to "`-x`".

2.14 Operator Precedence

In decreasing precedence:

operator	*associates*
array and field selection	Left
application	Left
`-` (unary)	Right
`^`	Right
`* /`	Left
`+ -`	Left
`to downto by`	-
`:`	Right
`++`	Right
`== <> < <= > >=`	Left
`and`	Left
`or`	Left
`,` (comma in tuples)	-

2.15 Voids

There is a special constant "`()`" whose type is `void`. There are no other values of this type. It is typically used in two situations: as an argument for a procedure that does not otherwise have a meaningful argument, and as a result of a procedure that does not otherwise have a meaningful result.

Although there is nothing non-functional about this by itself, it is useful mainly in non-functional programs (Sections 3, 4 and 5).

This is the second special use of parentheses, where they are not used for grouping (the first special use was to "quote" infix operators, Section 2.13).

2.16 Booleans

Booleans are defined as follows:

```
type    bool = False | True ;
typesyn B    = bool ;
```

with implicitly defined predicates:

```
False?    :: bool -> bool
True?     :: bool -> bool
```

Thus, `False` and `True` are identifiers representing boolean constants, and are also constructors (*i.e.,* they can be used in patterns).

Infix operators:

```
and :: bool -> bool -> bool
or  :: bool -> bool -> bool
```

Both left and right arguments are always evaluated.

The following operators are overloaded for booleans:

```
  ==    <>
```

See Appendix A.1 for standard boolean functions, including "`not`", the boolean negation function.

2.17 Integers

Integer numbers have type `int` (synonym: "`I`").

Integer constants are written as sequences of digits:

```
255 :: int
```

The following operators are overloaded for integers:

```
  - (unary negation)
  +    -    *
  ==   <>   <=   <   >   >=
```

Integers may also be used in patterns.

See Appendix A.2 for standard integer functions, including integer division.

2.18 Floats

Floating point numbers have type `float` (synonym: "`F`").

Floating point constants are written with decimal points and/or exponents:

```
0.6667 :: float
1.45   :: float
2.56e4 :: float
3e-3
```

The radix and exponent are always based on 10. The decimal point must be preceded or followed by at least one digit. The "`e`" must be preceded by a number and followed by a (possibly signed) integer. Example: `2.56e4` denotes 2.56×10^4

The following operators are overloaded for floating point numbers:

```
  - (unary negation)
  +    -    *
  ==   <>   <=   <   >   >=
```

Division:

```
/ :: float -> float -> float
```

Exponentiation:

```
^ :: float -> int -> float
```

See Appendix A.2 for standard floating point functions.

2.19 Characters

All characters have type **char** (synonym: "**C**"). The notation for character constants follows the conventions of the C programming language.

Most character constants are written as the character enclosed in single quotes:

```
'a'     :: char
'?'     :: char
'M'     :: char
...
```

To represent the single quote character itself, the backslash character and certain other characters, the following escape sequences may be used:

`'\n'`	::	**char**	newline
`'\t'`	::	**char**	horizontal tab
`'\v'`	::	**char**	vertical tab
`'\b'`	::	**char**	backspace
`'\r'`	::	**char**	carriage return
`'\f'`	::	**char**	form feed
`'\a'`	::	**char**	audible alert
`'\\'`	::	**char**	backslash
`'\?'`	::	**char**	question mark
`'\''`	::	**char**	single quote
`'\"'`	::	**char**	double quote
'*ooo*'	::	**char**	octal code
'\x*h..h*'	::	**char**	hex code

The last two forms are used for specifying octal or hexadecimal codes for characters. Octal character codes may be 1, 2 or 3 octal digit sequences. Hex character codes may contain any number of hexadecimal digits (≥ 1).

Upper- and lower-case are distinguished in character constants, except for hexadecimal digits in escape sequences.

The following operators are overloaded for characters:

`==   <>   <=   <   >   >=`

The obvious lexicographic ordering is guaranteed only within the following classes: digit characters, upper case characters, and lower case characters.

See Appendix A.3 for standard character functions.

2.20 Strings

All strings have type **string** (synonym: "**S**"). The notation for string constants follows the conventions of the C programming language.

String constants are written between double-quotation marks:

```
"Hiya"                    :: string
"Cab for hire"            :: string
"Wanna take you higher"   :: string
```

To represent the double quote character, the backslash character and certain other characters within strings, the same escape sequences as for character constants are available, with the following restriction: octal escapes must have exactly three octal digits, and hex escapes must have exactly two hexadecimal digits.

Upper- and lower-case are distinguished in string constants, except for hexadecimal digits in escape sequences.

Example: a string containing a newline.

```
"Some like it hot,\nSome like it cold."
```

The following operators are overloaded for strings:

`==   <>   <=   <   >   >=`

The ordering uses lexicographic ordering on the characters in the string.

Strings are zero-indexed (*i.e.,* the first character is at position 0).

The string type is different from lists or arrays of characters for reasons of efficiency.

Even though we use C notation for string constants, strings in Id are not zero-terminated as in C.

See Appendix A.4 for standard string functions, including functions to find the length of a string and to convert to and from lists and arrays.

2.21 Symbols

All symbols have type **symbol** (synonym: "**SYM**"). A symbol is written as a backslash followed by an identifier:

```
\A             :: symbol
\x'            :: symbol
\desmond_2_2   :: symbol
\c3po          :: symbol
\7am           :: symbol
```

The following operators are overloaded for symbols:

`==` `<>`

Unlike Lisp, symbols are are not related to program identifiers. Each distinct symbol merely represents a unique global constant (unique across all Id programs).

See Appendix A.5 for standard symbol functions, including conversion to and from strings.

2.22 Tuples

An N-tuple has type "`(t1,...,tN)`" where "`tj`" is the type of the j'th component.

Assuming:

```
e1     :: t1
...
eN     :: tN
```

then the tuple expression:

```
e1, ..., eN     :: t1,...,tN
```

denotes an n-tuple value (where $N \geq 2$—there is no notation for 1-tuples).

The comma has lower precedence than all other operators. Examples:

```
4+5, true           :: int,bool
5, (sqr x, false)   :: int,(int,bool)
(5,4),"Hi",(a > b)  :: (int,int),string,bool
```

The second expression is a 2-tuple whose second component is itself a 2-tuple. The nesting structure is significant—it is not equivalent to a 3-tuple.

Components of a tuple are accessed *via* pattern-matching (see Section 2.28).

2.23 Records

2.23.1 Record type definition

Records are like tuples with named fields. A new record type is defined using the statement:

```
type tx tv1 ... tvN =
                {record fieldname1 = t1;
                        ...
                        fieldnameM = tM};
```

Here, `tx` is the new type, and the `tvJ`'s are optional type variables ($N \geq 0$). The `fieldnameJ`'s are identifiers representing field names.

The type variables in the right-hand side, if any, must be a subset of the type variables mentioned in the left-hand side.

Fieldnames have the same syntax as program identifiers. However fieldnames and identifiers are drawn from different namespaces, *i.e.,* there can be an identifier called `x` and a fieldname called `x` with no confusion, since they always appear in disjoint regions of the program text.

The same fieldname may not be used in more than one record type (however, this restriction is likely to be removed in the future).

Examples:

```
type person = {record  pname = string ;
                       age   = int } ;

type complex = {record x = float;
                       y = float } ;

type node *0 = {record
                   nname    = int ;
                   info     = *0 ;
                   children = list (node *0)};
```

2.23.2 Record construction

A record is constructed using a `record` expression:

```
{record fieldname1 = e1;
        ... ;
        fieldnameM = eM}
```

The order in which the fields are specified does not have to follow the order of the fields in the record type definition. All fields must be specified (This is true only in the functional subset of the language; see Sections 4.4.2 and 5.4.2, where we allow I-structure and M-structure fields to be omitted).

Examples:

```
{record pname = "Z.Z.Gabor";
        age = 16 }            :: person

{record y = 2.3; x = 4.5 }    :: complex

{record nname = 2345;
        info = 6.001;
        children = Nil }    :: node float
```

2.23.3 Record field selection

Record fields may be selected using explicit field selection:

```
record_expression . fieldname
```

Examples:

```
p.pname

c.x

n.children
```

2.24 Conditional Expressions

Assuming:

```
e1  :: bool    e2  :: t    e3  :: t
```

then

```
if e1 then e2 else e3    :: t
```

is a conditional expression. The predicate `e1` is evaluated, and depending on its truth or falsity, either `e2` or `e3` (but not both) is evaluated, and returned as the value of the entire expression.

The conditional expression is equivalent to the following case expression (see Section 2.29):

```
{case e1 of
   True  = e2
 | False = e3}
```

In a conditional expression, the phrase "`else e3`" may be omitted:

```
if e1 then e2    :: void
```

This is syntactic shorthand for:

```
if e1 then e2 else ()
```

Although there is nothing non-functional about this by itself, it is useful mainly in non-functional programs (Sections 3, 4 and 5).

In parsing, an `else` matches the nearest preceding unbalanced `then`.

Precedence of `then` and `else`: binds less tightly even than commas—the parentheses may be omitted in each of these examples:

```
if ... else (x,y)
if ... then (f x y)
if ... else (x and y)
```

2.25 Blocks

The block expression:

```
{  STATEMENT ;
   ...
   STATEMENT
In
   e }
```

denotes the value of `e` evaluated in the environment inside the block.

Semicolons may be used as statement separators or as statement terminators, *i.e.,* the last statement before "`In`" is optionally followed by a semicolon.

Each statement must be well-typed. Statements usually specify *bindings* associating identifiers to types or values. The type of the block expression is the type of `e`.

Blocks (like all Id constructs) follow a static scoping discipline. The name-environment *inside* a block is the surrounding environment augmented by the names introduced by the statements of the block. A name `X` may be introduced at most once in a block, and hides any `X` in the surrounding environment. Names introduced inside a block are invisible outside the block.

Thus, the statements in a block may be recursive and mutually recursive, and the textual order of the statements is not significant.

The phrase "`In e`" may be omitted:

```
{ STATEMENT ;
  ...
  STATEMENT } :: void
```

This is syntactic shorthand for:

```
{ STATEMENT ;
  ...
  STATEMENT
In
  () }
```

Although there is nothing non-functional about this by itself, it is useful mainly in non-functional programs (Sections 3, 4 and 5).

2.26 Simple Binding Statements

The statement:

```
x = e ;
```

introduces `x` as a name for the value of expression `e` into the current scope. We also say that `x` is *bound* to the value of `e`.

2.27 Patterns

A *pattern* is one of the following:

- an identifier,
- an underscore "_" (don't care)
- a special constant (void, integer, character, symbol),
- a constructor pattern:
  `c pat1 ... patN`
  where `c` is an N-ary constructor name of some algebraic type `t`, and the `patJ`'s are themselves patterns ($N \geq 0$). The pattern itself is said to be of type `t`.

All normal identifiers in a pattern must be unique (technically, this is called left linearity). The "don't-care" pattern "_" may be repeated.

Special syntax: list patterns can be written with an infix colon:

```
pat1:pat2
```

Special syntax: N-tuple patterns can be written with commas:

```
pat1,...,patN
```

2.28 Pattern-Matching

Pattern-matching a pattern to a value has two distinct aspects:

- *Matching*, *i.e.,* testing whether the value is a data structure that conforms to the shape specified by the pattern. If so, the match is said to succeed, otherwise it is said to fail.
- *Binding*, *i.e.,* producing an environment in which identifiers in the pattern are bound to corresponding components of the value. Binding only occurs if the match succeeds.

Matching:

Matching succeeds under the following conditions.

A "don't-care" pattern "_" successfully matches any value.

An identifier pattern `x` successfully matches any value.

A constant pattern `c` successfully matches only the corresponding value c.

A constructor term pattern

```
c pat1 ... patN
```

successfully matches only a value of the form

```
c v1 ... vN
```

provided also that each `patJ` matches the corresponding field `vJ`.

Binding

Assuming the pattern matches successfully, the following environment is produced:

A "don't-care" pattern "_" produces no binding.

An identifier pattern `x` binds `x` to the value.

A constant pattern `c` produces no binding.

A constructor term pattern:

```
c pat1 ... patN
```

when matched to a value:

```
c v1 ... vN
```

produces the union of all the bindings obtained by matching all the `patJ`'s to their corresponding `vJ`'s.

2.29 Case-expressions

Assuming:

```
e  :: te
e1 :: t
...
eN :: t
```

and `pat1` ... `patN` are patterns of type `te`, then the case-expression:

```
{case e of
   pat1 = e1
|  ...
|  patN = eN } :: t
```

behaves as follows. All the patterns `pat1 ... patN` are matched to the value of `e`, in no specific order. No more than one match should succeed. If `patJ` succeeds, then the resulting bindings augment the current environment, eJ is evaluated in that environment, and its value is returned as the value of the whole expression. Note that all eJ's must have the same type t, and the entire case-expression has type t.

The patterns must be disjoint, *i.e.,* at most one pattern can successfully match any e; this is checked by the compiler. The pattern-matching is "parallel"— there is no specified top-to-bottom or left-to-right order in the pattern-matching. (For specialists: disjoint patterns may require non-sequential functions for non-strict evaluation; in Id, these become strict so that the order of patterns is not significant).

The last clause may be preceded by ".." to designate it as a catch-all clause (this is a limited form of ordering):

```
{case e of
    pat1 = e1
|   ...
|   patN = eN
|.. patF = eF }
```

Here, a match of patF to v (the value of e) is attempted only if all other matches fail. Thus, patF need not be disjoint from the other patterns.

The patterns need not be exhaustive—if no pattern matches, a runtime error occurs, and the case-expression has the error value $\perp$ (see Section 2.38). This behavior can be understood as follows. A case expression without a default clause is equivalent to one with a default clause added:

```
{case e of
    pat1 = e1
|   ...
|   patN = eN
|..    _  = error "Pattern match"}
```

(see Section 2.38 for the error function). A case expression with a default clause is equivalent to one with the default clause rewritten as follows:

```
{case e of
    pat1 = e1
|   ...
|   patN = eN
|..    x  = {case x of
                patF = eF
           |..     _  = error "Pattern match"}}
```

2.30 Function Abstractions

A function abstraction expression (a form of lambda-expression) is written:

```
{fun pat11 ... pat1N = e1
|    pat21 ... pat2N = e2
     ...
|    patM1 ... patMN = eM
|..  patL1 ... patLN = eL }
```

and is equivalent to:

```
{fun x1 ... xN =
    {case (x1,...,xN) of
        (pat11,...,pat1N) = e1
    |   (pat21,...,pat2N) = e2
        ...
    |   (patM1,...,patMN) = eM
    |.. (patL1,...,patLN) = eL }}
```

and represents an "anonymous" function of arity N (≥ 1) whose formal parameters are the xJs and whose body is the case-expression. As usual, static scoping rules are followed.

The final "catch-all" clause (signalled by "..") is optional.

2.31 Function Definitions

A function definition statement is written:

```
def   f pat11 ... pat1N = e1
 |    f pat21 ... pat2N = e2
        ...
 |    f patM1 ... patMN = eM
 |..  f patL1 ... patLN = eL
```

and is equivalent to the simple binding statement:

```
f = {fun x1 ... xN =
        {case (x1,...,xN) of
            (pat11,...,pat1N) = e1
        |   (pat21,...,pat2N) = e2
        |   ...
        |   (patM1,...,patMN) = eM
        |.. (patL1,...,patLN) = eL }}
```

The final "catch-all" clause (signalled by "..") is optional.

See also Section 7.1 for defsubst, a version of def that also suggests that it is inlinable for efficiency.

2.32 Pattern-Binding Statements

A convenient way to access components of a data structure is to use a pattern-binding statement:

```
pat = e ;
```

The pattern is matched against the value of `e`, and if it succeeds, the corresponding bindings are introduced into the current scope.

If the match fails, a runtime error occurs—the statement is equivalent to the error statement $statement_\perp$ (see Section 2.38), and some or all of the bindings may not be performed.

Example:

```
(x,y:ys) = e ;
```

The expression `e` should evaluate to a 2-tuple whose second component is a non-empty list. The statement binds `x` to the first component of the tuple, `y` to the head of the list and `ys` to the tail of the list. If the second component of the 2-tuple is an empty list, the pattern-match fails; in this case, the statement is equivalent to the error statement $statement_\perp$ (see Section 2.38), `y` and `ys` remain unbound, and it is unspecified whether `x` is bound to the first component of the tuple or remains unbound.

2.33 Lists

The standard list type is defined as:

```
type list *0 = Nil | Cons *0 (list *0);
```

with implicitly defined field selectors:

```
Cons_1    :: (list *0).*0
Cons_2    :: (list *0).(list *0)
```

and implicitly defined predicates:

```
Nil?    :: (list *0) -> bool
Cons?   :: (list *0) -> bool
```

Special syntax: constructor terms of the form:

```
Cons e1 e2
```

may be written with the infix ":" operator:

```
e1:e2
```

and the higher-order function:

```
Cons
```

may be written as:

```
(:)
```

using the usual parenthesized notation for quoting infix operators. Also, the pattern:

```
Cons pat1 pat2
```

can be written

```
pat1 : pat2
```

2.33.1 Binary Infix List Operators

Appending two lists:

```
++ :: (list *0) -> (list *0) -> (list *0)
```

2.33.2 Arithmetic Series Operators

Assuming:

```
e1   :: int    e2  :: int    eInc  :: int
```

evaluate to integers $v1$, $v2$ and $vInc$, respectively, then the expressions:

```
e1 to e2 by eInc         :: (list int)
e1 downto e2 by eInc     :: (list int)
```

produce lists containing ($v1$, $v1 + vInc$, $v1 + 2vInc$, ..., $v2$), and ($v1$, $v1 - vInc$, $v1 - 2vInc$, ..., $v2$), respectively.

Note: $vInc$ must always be positive.

The short forms:

```
e1 to e2     :: (list int)
e1 downto e2 :: (list int)
```

assume that $vInc$ is $+1$.

Precedence of `to`, `downto` and `by`: the parentheses may be omitted in each of these examples:

```
... to (f x)
... downto (f x)
... to (e1 + e2)
... by (f x)
```

See also Section 6.2.1 for arithmetic series of infinite length.

2.33.3 List Comprehensions

A list-comprehension is written:

```
{: e || gen1 ; ... ; genN }
```

($N \geq 1$). Each generator `gen` is written in one of two ways:

```
pat <- eList FILTER1 ... FILTERm
pat  = eVal  FILTER1 ... FILTERm
```

($m \geq 0$). Each `FILTER` is written in one of two ways:

```
when   epw
unless epu
```

Individual generator behavior
In the first form (using `<-`), `eList` must be a list of values; `pat` is matched to each element of the list, generating a sequence of environments that bind the pattern variables. If pattern-matching fails, a runtime error occurs— the failing match of the pattern to a list element is replaced by the error statement $statement_\perp$ (see Section 2.38).

In the second form (using `=`), `pat` is matched to the value of `eVal`, generating an environment that binds the pattern variables. If pattern-matching fails, a runtime error occurs— the failing match of the pattern to a list element is replaced by the error statement $statement_\perp$ (see Section 2.38).

The environments are then filtered, *i.e.,* those environments in which an `epw` evaluates false or an `epu` evaluates true are discarded. The filters are tried in sequence from left to right, *i.e.,* if a filter rejects an environment, the subsequent filter expressions are not evaluated for that environment.

Generator sequence behavior
The generators are evaluated from left to right. For each environment Env in the sequence of environments produced before `genJ`,

- `genJ` is evaluated in Env, and produces a set of environments Env_{J1}, Env_{J2}, ...
- Env is replaced in the sequence by the augmented environments $Env + Env_{J1}$, $Env + Env_{J2}$, ...

Thus, the net result of the generator sequence is a sequence of environments containing bindings for the pattern variables of all the generators.

List-comprehension behavior
The expression `e` is evaluated in each environment produced by the generator sequence, and the values are collected into a list (in the same order), which is the result of the whole expression.

Type of list comprehension
The type of the list comprehension is `(list t)`, where `e::t` in the environment specified by the generators.

Examples

A list of x-y coordinates in the first octant of a 100-square:

```
{: x,y || x <- 0 to 100 ; y <- 0 to x }
```

A list of x-y coordinates in a 100-square that are not on the axes or on the diagonals:

```
{: x,y || x <- 0 to 100 when x <> 0
        ; y <- 0 to 100 when y <> 0
                        unless x == y }
```

See Appendix A.6 for standard list functions.

2.34 Arrays

Arrays are collections of uniformly-typed objects, with a constant access-time for each component.

2.34.1 Array Types

An n-dimensional array ($n \geq 1$) whose components have type `t` has type:

```
nD_array t
```

Arrays can contain objects of any type, including other arrays.

Nested arrays are not equivalent to multi-dimensional arrays. The following two types are not equivalent:

```
2D_array t
```

```
1D_array (1D_array t)
```

Synonyms for `1D_array`:
`vector` `array`

Synonym for `2D_array`:
`matrix`

2.34.2 Array literals

Assuming a bounds expression (an n-tuple of integer 2-tuples):

```
eBounds :: (int,int), ... ,(int,int)
```

which evaluates to (l1,u1),...,(ln,un), an array with those index bounds can be constructed by enumerating its contents:

```
{nD_array eBounds of
 eFirst  ...  eLast}
```

The expressions specify the contents in "row-major" order, *i.e.,* starting from index l1,l2,...,ln and ending at index u1,...,un, stepping the right-most index fastest. The number of contents-expressions must be equal to the number of components in the array.

Note that if a component is specified by an application "f a", it will have to be parenthesized to prevent it from being mistaken as two separate component specifications.

It is legal for the index bounds (l, u) along any dimension to be empty, *i.e.,* to have $u = l - 1$ (zero components).

2.34.3 Array Selection

Assuming:

```
a    :: nD_array t
e    :: int,...,int
```

then the array-selection expression:

```
a[e]    :: t
```

returns the value of the j1,...,jn'th component of the array "a", where j1,...,jn is the value of "e". The index j1,...,jn must be within the index-bounds of the array—it is a runtime error otherwise, producing the error value ⊥ (see Section 2.38).

Note that the index expression can be *any* expression that returns an n-tuple of integers, *i.e.,* it does not have to be a literal tuple-expression.

Precedence of array selection: tighter even than application, so that parentheses are not necessary in:

```
f (a[e])
```

2.34.4 Array Index Bounds

For each $n \geq 1$, there is a function that returns the index bounds of n-dimensional arrays:

```
bounds~1D_array ::
    (1D_array *0) -> (int,int)
bounds~2D_array ::
    (2D_array *0) -> ((int,int),(int,int))
...
```

The overloaded identifier bounds represents all these functions.

2.34.5 Array Comprehensions

Array comprehensions are used to construct arrays, allowing the programmer to specify the contents of different regions of the array using different computation rules.

For each $k \geq 1$ and $n \geq 1$, assuming a bounds expression (an n-tuple of integer 2-tuples):

```
eBounds :: (int,int), ... ,(int,int)
```

and a set of index expressions (each an n-tuple of integers):

```
eJ1 :: int,...,int
```

and a set of component expressions (each a k-tuple):

```
eJ2 :: t1,...,tk
```

then the array comprehension expression:

```
{k_nD_arrays eBounds of
  [e11] = e12 || gen ; ... ; gen
| ...
| [eM1] = eM2 || gen ; ... ; gen }
```

returns a k-tuple of n-dimensional arrays. Each gen is a generator, possibly including filters (exactly as in list-comprehensions).

Array comprehension behavior

eBounds is evaluated to produce lower- and upper-bounds for each of n dimensions. k arrays with these dimensions are created. Then, the subsequent clauses are all executed in parallel to fill the arrays. (The top-to-bottom order of the clauses has no significance.)

Clause behavior

See *Generator behavior* and *Generator sequence behavior* in Section 2.33.3 on list comprehensions to see how each generator sequence

```
gen ; ... ; gen
```

produces a sequence of environments. Now, in each such environment, eJ1 is evaluated to produce an index into the arrays (an n-tuple), and eJ2 is evaluated to produce a k-tuple specifying the contents of that location in each of the k arrays.

Type of array comprehensions

The type of the value of the array comprehension is

```
nD_array t1, ..., nD_array tk
```

where `t1,...,tk` is the type of each `eJ2` in each environment specified by the J'th generator sequence.

A runtime error occurs if the contents of an array at some index is defined more than once, *i.e.,* if the array comprehension specifies values twice at the same index `j1,...,jN`. This is a drastic error—the entire program is considered to be inconsistent (see Section 2.38).

If, at some index, the array comprehension specifies no value at all, then that location simply remains undefined (indistinguishable from a non-terminating computation).

The generator sequences "`|| gen ; ... ; gen`" are optional. In this case, `eJ1` and `eJ2` specify the contents of a single location.

The `k_nD_arrays` keywords have synonyms for some common cases:

	$k=1$	$k\geq 1$
$n=1$	`array` `vector`	`k_arrays` `k_vectors`
$n=2$	`matrix`	`k_matrices`
$n\geq 1$	`nD_array`	

It is legal for the index bounds (l,u) along any dimension to be empty, *i.e.,* to have $u=l-1$ (zero components).

Examples

The vector sum of two vectors `A` and `B`:

```
{array (1,N) of
  [i] = A[i]+B[i] || i <- 1 to N }
```

A statement defining an array using a "wavefront" recurrence:

```
A = {matrix (1,N),(1,N) of
       [1,1] = 1
     | [i,1] = 1              || i <- 2 to N
     | [1,j] = 1              || j <- 2 to N
     | [i,j] = A[i-1,j] +
               A[i,j-1]       || i <- 2 to N
                               ; j <- 2 to N };
```

An array containing the inverse of a given permutation in array `A`:

```
{array (1,N) of
   [A[i]] = i || i <- 1 to N }
```

See also Appendix A.8 for standard array functions.

2.35 Accumulators

Accumulators are an extension of arrays.

```
{k1_n1D_arrays eBounds1 of
   [ei] = evs || gen ; ... ; gen
 | ...
 | [ei] = evs || gen ; ... ; gen
 k2_n2D_arrays eBounds2 of
   [ei] = evs || gen ; ... ; gen
 | ...
 | [ei] = evs || gen ; ... ; gen
 .
 .
 .
 kM_nMD_arrays eBoundsM of
   [ei] = evs || gen ; ... ; gen
 | ...
 | [ei] = evs || gen ; ... ; gen

 accumulate k_ops of

   [eis] = evs || gen ; ... ; gen
 | ...
 | [eis] = evs || gen ; ... ; gen }
```

This returns a tuple containing k arrays ($k = k1 + k2 + ... + kM$). The first $k1$ arrays have bounds `eBounds1` and are initialized according to the first set of clauses, the next $k2$ arrays have bounds `eBounds2` and are initialized according to the second set of clauses, and so on.

After the keyword `accumulate`, the expression `k_ops` returns a tuple of k operators that are the accumulation operators.

The final set of clauses specifies the indexes and values for the accumulation. In each clause, `eis` is a k-tuple of indices `i1,...,ik`, and `evs` is a k-tuple of values `v1,...,vk`, specifying the accumulations:

```
X1[i1] := op1 X[i1] v1
...
Xk[ik] := opk X[ik] vk
```

Each such accumulation is executed atomically.

The array value of the entire expression is returned only after *all* the accumulations have been done. Thus, accumulators are hyperstrict, unlike ordinary arrays, which are non-strict.

Since the order in which the accumulation operations are performed is non-deterministic, it is the programmer's responsibility to ensure that the accumulation operators have the following property:

```
(op (op x y) z) = (op (op x z) y)
```

so that the whole construct is deterministic.

Example

A 10-category histogram of a zillion things:

```
{array (1,10) of
  [i] = 0 || i <- 1 to 10
 accumulate (+) of
  [classify x] = 1 || x <- zillion_things }
```

2.36 Abstract Types

A new abstract data type is declared using this statement:

```
abstype NEWTYPE
 typeof x1 = TYPE1 ;
 ...
 typeof xN = TYPEN
 rep
   REPRESENTATION-TYPE
   {
     ...
     def x1 = ... ;
     ...
     def xN = ... ;
     ...
   };
```

`NEWTYPE` is the (possibly parameterized) new type expression.

The subsequent `typeof` statements specify the *signature* (or interface) of the abstract type.

The `REPRESENTATION-TYPE` is a type-expression specifying the internal representation of objects of the new type.

The statements in the braces specify definitions for the identifiers in the signature. There may be other identifiers defined in the braces, but they are not exported—they are just local types and local definitions.

The net effect of the `abstype` statement is to introduce the new type identifier and the identifiers `x1` through `xN` into the current scope. Each `xJ` has the specified type signature and bound value.

Within the braces, the abstract type is treated as equivalent to the representation type. Outside the `abstype` statement, the abstract type and the representation type are treated as distinct (different) types.

The representation type may be a local type, *i.e.,* declared in the braces. If so, it is not even visible outside.

Example

A stack, with a list representation:

```
abstype (stack *0)
  typeof empty = (stack *0);
  typeof empty?= (stack *0) -> bool;
  typeof push  = *0 ->
                 (stack *0) -> (stack *0);
  typeof pop   = (stack *0) -> (stack *0);
  typeof top   = (stack *0) -> *0
 rep (list *0)
 {
   empty = nil ;
   empty? = nil? ;
   push = (:) ;

   def pop (x:s) = s
     | pop nil   = error "Stack underflow" ;

   def top (x:s) = x
     | top nil   = error "Stack underflow"
 };
```

(See Section 2.38 for the `error` function.)

2.37 Loops

While list- and array-comprehensions are convenient for expressing "mapping" operations over sequences, loops are convenient for expressing "reduction" operations. Id has while-loops and for-loops.

The general while-loop expression notation is:

```
{while eb do
   STATEMENT ;
```

```
   ...
   STATEMENT
 finally e}    :: t
```

where `eb :: bool` and `e :: t`.

The general for-loop expression notation is:

```
{for x <- eIndex do
   STATEMENT ;
   ...
   STATEMENT
 finally e}    :: t
```

where `eIndex :: (list *0)` and `e :: t`. The for-loop notation is shorthand for a while-loop:

```
{ L = eIndex
In
  {while (L <> nil) do
     x:(next L) = L ;
     STATEMENT ;
     ...
     STATEMENT
   finally e}}
```

Here, `eIndex` is normally an arithmetic-series expression (see Section 2.33.2).

The braces are compulsory. The type of the entire loop expression is the type of the expression in the "`finally`" phrase.

The loop body is a series of statements, with the following extension: a binding occurrence of an identifier (say "`x`") may be prefixed by the keyword "`next`", denoting the value to be used for "`x`" in the next iteration. This value is also available in the current iteration because "`next x`" may be used as an expression.

The phrase "`finally e`" may be omitted:

```
{while/for ...
  STATEMENT
  ...
  STATEMENT } :: void
```

This is syntactic shorthand for:

```
{while/for
  STATEMENT ;
  ...
  STATEMENT
finally () }
```

Although there is nothing non-functional about this by itself, it is useful mainly in non-functional programs (Sections 3, 4 and 5).

See Section 7.2 on "bounded loops" for annotations to limit the parallelism of loops.

2.37.1 Scope of Variables in Loops

We use the phrase *loop context* to refer to the set of variables available to the loop expression from the surrounding scope. Any variable "`x`" from the loop context takes on a new value at each iteration if there is a "`next x`" binding in the loop body.

In `while`-loops, the predicate may only use identifiers from the loop context, and is re-evaluated each time *before* entering the loop body.

The loop is *terminated* in `while` loops when the predicate evaluates to `false`. Then, the `finally e` expression is evaluated and returned as the value of the loop. It may only use identifiers from the loop context.

Within the loop body, only variables from the loop context may be "nextified". The loop body may also contain ordinary identifier bindings. The scope of all bindings is the entire loop body (this includes the nextified variables, since "`next x`" may be used as an expression within the body).

For any nextified identifier "`x`", the bound value becomes the value of "`x`" at the end of the iteration.

2.37.2 Loop semantics

While-loops are equivalent to tail-recursive functions, according to the correspondence illustrated by the following example. Assume that `x` and `y` are the only two nextified variables, and that their bindings are the last two statements in the loop body:

```
{while eB do
   STATEMENT
   ...
   STATEMENT
   next x = ex ;
   next y = ey ;
 finally eF}    :: t
```

This loop is equivalent to:

```
{ def loop x y = if eB then
                    { STATEMENT
                      ...
                      STATEMENT
                      next_x = ex ;
                      next_y = ey ;
                    In
                      loop next_x next_y}
                 else
```

```
                    eF ;
In
  loop x y } ;
```

The function `loop` has a parameter for each nextified variable. The loop body is transcribed verbatim into the block inside the conditional, with the exception that we systematically replace all occurrences of "`next x`" by the identifier `next_x`. The return-expression of the block is a tail-recursive call to `loop` using the next values of `x` and `y`.

The translation illustrates a number of points. The initial values of `x` and `y` come from the surrounding scope. If `eB` evaluates to `False` the first time, the loop body is not executed at all. Because of the parallel semantics of blocks, the recursive call (to the next iteration) may execute in parallel with the current call. In principle, *all* iterations may execute in parallel and, indeed, the loop can even return a final result while the loop bodies are still executing.

Bounded Loops

The above description of loops semantics is correct to first order. However, the user may specify that a loop should be compiled as a bounded loop, which limits the degree of unfolding to a fixed number of iterations. The net effect of bounded loops is possibly to increase the strictness of loops in exchange for better resource utilization, *i.e.,* it can affect termination. Bounded loops are discussed fully in Section 7.2.

In the absence of explicit user directives, it is unspecified whether loops are compiled as bounded loops or not.

Examples

Successive approximations until convergence to a limit:

```
{ approx = first_guess ;
  delx   = infinity
In
  {while (delx > epsilon) do
     next approx = improve approx ;
     next delx   = (next approx)-approx
   finally x}}
```

The n'th Fibonacci:

```
{ x,y = 1,1
In
  {for j <- 1 to n do
     next x,next y = y, x+y
   finally x}}
```

2.38 Errors

There are three types of errors that may be produced during execution of an Id program:

- Error values, written $\perp$. For example, the expression `v1/v2` evaluates to $\perp$ if `v2` is zero. Error values should be regarded as synonymous with non-terminating computations, *i.e.,* it is as if the erroneous operation that produced $\perp$ is stuck forever.
- Error statements, written $statement_\perp$. For example, a pattern-binding statement:

  ```
  (x:y) = e ;
  ```

  is replaced by $statement_\perp$ if `e` evaluates to `Nil` (*i.e.,* the pattern-match fails). Another reason for this error is an out-of-bounds index in an array-store operation (see Sections 4 and 5).
- Inconsistency, written $\top$. This is a drastic error, in that it renders the entire program inconsistent. For example, in an array comprehension, if some element of the array is specified more than once, the program is inconsistent. The usual reason for this error is multiple I-stores into the same location (see Section 4).

For $\perp$ and $statement_\perp$, implementations of Id may provide some interactive means for the user to artificially repair the error and continue execution. For $\top$, on the other hand, no such repair is meaningful.

Functional Id programs and programs that use I-structures (Section 4) are determinate (Church-Rosser) even in the presence of errors. A program:

- may be inconsistent, or
- produces a value (perhaps the error value $\perp$) and zero or more $statement_\perp$s.

This behavior is repeatable, *i.e.,* different runs of the program will always produce the same outcome, despite differences in runtime scheduling.

Forced errors

The programmer can force an error using the function:

```
error      :: string -> *0
```

which always evaluates to $\perp$. The argument string `e` should be a meaningful error-message. Because of its polymorphic output type, the `error` function may be applied in any context.

3 General issues concerning non-functional constructs (I-structures and M-structures)

I-structures are a small departure, and M-structures are a major departure from purely functional semantics. I-structures and M-structures are layered on top of the purely functional subset of Id by constructs that are distinguished by syntax and by type, *i.e.,* it is possible to mechanically check whether a program is purely functional or whether it uses I-structures or M-structures.

This section discusses general issues concerning these non-functional extensions. I-structures are described in detail in Section 4, and M-structures are described in detail in Section 5.

3.1 I-structure and M-structure semantics

The primitive side-effecting constructs in Id have to do with updating components of data structures. There is no assignment statement for ordinary variables. A component of a data structure may have either functional, I-structure or M-structure semantics. Operations on all three have built-in synchronization.

The value of a functional component of a data structure is specified simultaneously with the creation of the data structure. Of course, the component cannot be updated, and it can be read many times. The programmer never considers synchronization explicitly, except inasmuch as one is aware that one can use non-strictness to define data structures using recurrences. All data structures described in Section 2 were functional.

For a non-functional component of a data structure, no value is specified when the data structure is created; instead, a separate assignment statement is used. A non-functional component may be in one of two states: *full* (with a value), or *empty*. All components begin in the empty state (when the data structure is allocated) and later become full through assignment statements. I-structure and M-structure components have different semantics for reading and writing, and are described in detail in Sections 4 and 5, respectively.

For algebraic types and records, different fields of an object may have different semantics: functional, I-structure or M-structure. The semantics of each field is specified in the type declaration. A given field can only be accessed according to its declaration—with functional, I-structure or M-structure semantics; this is ensured by a combination of syntax and type checking.

For arrays, on the other hand, there are three types of arrays: functional, I-arrays and M-arrays. Thus, all components of an array have the same semantics, and this is reflected in the array type itself. Again, type-checking ensures proper access.

3.2 Polymorphism of I-structures and M-structures

Updatable structures do not mesh well with polymorphism. In order to be safe, our type-checker is very conservative, and so the polymorphism of some programs with I-structure objects may be less than expected.

(Comment for experts: the polymorphism of I-structure and M-structure components is expressed with so-called "weak" type variables, similar to ref types in ML.)

3.3 Referential Transparency, Sharing and Object Identity

Programs that use only functional operations are referentially transparent, whereas programs that use I-structures or M-structures may not be. For example, consider the following two expressions:

```
(e,e)                   { x = e
                          In
                            (x,x) }
```

These expressions are equivalent (except for efficiency considerations) in the functional subset of Id, *i.e.,* both programs will produce exactly the same answer except that one may use more resources than the other. They are no longer equivalent if they use I-structures or M-structures. For example, if `e` allocates and returns a data structure, the expression on the left produces two references to two separate data structures, whereas the expression on the right produces two references to the same structure. In the functional subset of Id, these two situations are indistinguishable. With I-structures or M-structures, on the other hand, they are distinguishable, because an assignment *via* one reference to a data structure can affect what is read *via* another reference to the same data structure.

Thus, when programming with I-structures and M-structures, the programmer should be clear about the sharing of computations and, by extension, the sharing of data structures. The standard procedure:

```
same? :: *0 -> *0 -> bool
```

can be used to test if two values are identical (the same object). Its behavior is undefined on non-updatable objects, such as numbers, functions and algebraic types with no updatable components. For example,

```
same? (5,6) (5,6)
```

may return true or false, depending on the implementation. The programmer is advised to use this procedure only on updatable objects (I-arrays, M-arrays, and algebraic types and records with I-fields or M-fields).

3.4 What gets evaluated, and when

Consider a conditional expression:

```
if e1 then e2 else e3
```

In Section 2.24, we stated that `e1` is evaluated first and then, depending on its value, either `e2` or `e3` (*but not both*) is evaluated.

In the functional subset of Id, this level of precision is not necessary— neither *what* nor *when*. Even if some evaluation occurs in *both* `e2` and `e3`, some unnecessary work would be done, but it would not affect the outcome of the program. Further, it does not matter if some evaluation occurs in `e2` or `e3` before `e1` is evaluated.

With I-structures and M-structures, however, both these issues are important. For example, if `e2` and `e3` both contained assignments to the same I-structure location, a runtime error would occur if both were performed. Or, if `e2` or `e3` manipulated some M-structure location, they could affect each other's outcome, or even the outcome of other expressions such as `e1`, if they were evaluated too early.

Thus, when using I-structures and M-structures, the programmer must be clear about exactly which expressions get evaluated, and when.

In general, for most expressions that have multiple sub-expressions, all sub-expressions are evaluated, and they are evaluated in parallel. For example, all expressions in a block, both expressions in an infix expression "`e1` *op* `e2`", all expressions in a tuple expression, *etc.* are evaluated in parallel. The exceptions to this general rule are described below.

Conditionals:

```
if e1 then e2 else e3
```

The predicate `e1` is evaluated fully. After its boolean value is available, one of the arms `e2` or `e3` is evaluated. However, please note the following subtlety due to non-strictness: this does not mean that there is no overlap between the evaluation of `e1` and the evaluation of `e2/e3`. Consider this conditional:

```
if (Nil? (eH:eT)) then e2 else e3
```

Because of non-strictness, the predicate can return `False` even if no evaluation of `eH` and `eT` has yet taken place, and this enables the evaluation of `e3`. Thus, the evaluations of `eH`, `eT` and `e3` can overlap. Thus, non-strictness should be kept in mind when reasoning about *when* an expression is evaluated, in this and all subsequent rules.

Case expressions (this is the general form of the rule for conditionals):

```
{case e of
   pat1 = e1
|  ...
|  patN = eN }
```

The expression `e` is evaluated completely. When it is successfully matched to one of the patterns, say `patJ`, the corresponding arm `eJ` is evaluated completely.

Function definitions and applications:

```
def f x1 ... xN = eBody ;
```

Nothing is evaluated in `eBody`. After the function `f` has been applied to N arguments, a new instance of `eBody` is created and is evaluated. Partial applications of `f` (*i.e.,* application to fewer than N arguments) do not cause any evaluation in `eBody`.

Function abstractions have similar behavior. Evaluating the expression:

```
{fun x1 ... xN = eBody}
```

does not cause any evaluations in `eBody`— it just creates a function value (a closure). Partial applications of this function value (*i.e.,* applications to fewer than N arguments) simply build new function values. When fully applied to N arguments, a new instance of `eBody` is created and is evalauted.

Loops: the evaluation order is derived from the translation to tail-recursive form, as described in Section 2.37.2. Briefly, without going into this translation, in a while-loop:

```
{while eb do
   STATEMENT ;
   ...
   STATEMENT
 finally e}
```

none of the statements in the body are executed until `eb` evaluates to `True`, after which all the statements and the next invocation of `eb` are evaluated in parallel. After a particular invocation of `eb` evaluates to `False`, the corresponding loop body is not executed at all, and the corresponding final expression `e` is evaluated.

Delayed expressions: See Section 6

Barriers: See Section 3.7.

3.5 Determinacy

Programs in the functional subset of Id are guaranteed to be determinate, by which we mean that that it is impossible for the programmer to write a program that, despite different schedules on different runs, produces two different outcomes. Formally, functional programs are said to have the Church-Rosser property.

Programs which use only functional data structures and I-structures (not M-structures) are also guaranteed to be determinate (despite the loss of referential transparency).

Programs that use M-structures are not guaranteed to be determinate. Of course, through careful use of M-structures, it is still possible to write determinate programs (indeed, this is often an explicit goal), but it is important to understand the difference: in functional Id and with I-structures, determinacy is a property of the *language* (every program has this property), whereas with M-structures, determinacy is only a property of particular programs (has to be proved separately for each program).

3.6 Side-effect statements

In the functional subset of Id, a statement always binds identifiers on the left-hand side to values from the right-hand side (*e.g.,* in a block or loop body). With I-structures and M-structures, we can also have *side-effect* statements.

Primitive side-effect statements are assignments of values to I-structure or M-structure components. Such a statement has the form:

```
slot-designator = e ;
```

where `slot-designator` is an identifier (not a general expression) followed by zero or more functional or I-structure array and field selectors, followed by exactly one I-structure or M-structure array or field selector. Examples:

```
a[3] = e1 ;
b![4] = e2
c.name = e3;
d!balance = e4;
e[4].name = e5 ;
f.months[12]!balance = e6 ;
```

Note that slot-designators are unrelated to patterns, and there are no identifier bindings involved.

An expression `e` can be executed purely for its side effect by using it in a statement that discards its result:

```
STATEMENT ;
...
_ = e ;
...
STATEMENT
```

The single underscore may be regarded as a special, dummy identifier to which the value of `e` is bound and never referred to further.

However, conditionals, loops, case and block expressions can be used directly as statements (the "_ = " can be omitted):

```
STATEMENT ;
...
if ... then ... else ;
{while/for ...    } ;
{ STATEMENT; ... ; STATEMENT in e'} ;
{case ...    } ;
...
STATEMENT
```

3.7 Sequencing Statements: barriers

By using "---" in a statement sequence, the programmer can indicate that all statements before it must execute completely before any statement after it can begin:

```
{ s1 ;
  ...
  sI ;
  ---
  sJ ;
  ...
  sN
In
  e }
```

Here, `s1` through `sI` execute completely before the execution of `sJ` through `sN` or `e` begins.

When we say that a statement S must "execute completely", we include, transitively, anything that S calls, anything that those computations in turn call, and so on. This is sometimes also referred to as "hyperstrict" evaluation. The "---" lexeme can be read visually as a "barrier".

Unless the last statement in a block is followed by an "---", it is executed in parallel with the return-expression. Compare these two expressions:

```
  { s1                  { s1
    ---                   ---
    s2                    s2
  In                      ---
    e}                  In
                          e }
```

On the left, `e` is evaluated in parallel with `s2`; on the right, `e` is evaluated after the execution of `s2`.

Parentheses may be used to group statements to limit the extent of the sequentialization:

```
{ s1 ;
  ( s2
    ---
    s3 ;
    s4 ) ;
  s5
In
  e }
```

Here, `s1`, the parenthesized statement group, `s5` and `e` are evaluated in parallel. Within the statement group, `s2` is evaluated first, after which `s3` and `s4` are evaluated in parallel.

Grouping statements using parentheses does not affect the scope of identifiers (hence one could not in general replace the parentheses by braces in the block shown above). For example:

```
{ s1 ;
  ( x = .... y ... ;     % s2
    ---
    s3 ;
    s4 ; ) ;
  y = e5 ;
In
  e }
```

The use of y in statement s2 is perfectly legal; there is no violation of scope rules.

Similarly, barriers and statement groupings, which concern dynamic control, are irrelevant for type definitions and type declarations, since these are static declarations. For uniformity, the programmer may regard all type definitions and declarations in a block as if they occurred at the top of the block.

However, the programmer should notice that while the order of the statements has not effect on the scopes of identifiers, the order can be important to prevent deadlock. For example:

```
{ s1 ;
  ( x = .... y ... ;     % s2
    ---
    s3 ;
    y = e4 ; ) ;
  s5 ;
In
  e }
```

There is no violation of scope rules, but the program will deadlock because statement s2 cannot complete until y gets a value, and y cannot get a value until s2 completes.

The behavior of barriers in loop bodies follows from the translation to tail-recursive functions described in Section 2.37.2. Consider the following loop:

```
{while eB do
   STATEMENT1;
   ---
   STATEMENT2;
   next x = ex ;
   next y = ey ;
 finally eF}
```

The translation is:

```
{ def loop x y = if eB then
                   { STATEMENT1;
                     ---
                     STATEMENT2;
                     next_x = ex ;
                     next_y = ey ;
                   In
                     loop next_x next_y}
                 else
                    eF ;
In
  loop x y } ;
```

From this, it is obvious that the recursive call for the next iteration cannot begin until STATEMENT1 completes. The net effect is that the entire loop runs sequentially. The programmer should be careful about this behavior: *a single unadorned barrier in a loop body sequentializes the loop!*

If the programmer wishes to localize the barrier to individual iterations while allowing separate iterations to run in parallel, parentheses may be used as usual to localize the barrier. For example:

```
{while eB do
   ( STATEMENT1;
     ---
     STATEMENT2; )
   next x = ex ;
   next y = ey ;
 finally eF}    :: t
```

The translation is:

```
{ def loop x y = if eB then
                   { ( STATEMENT1
                         ---
                       STATEMENT2 )
                     next_x = ex ;
                     next_y = ey ;
                   In
                     loop next_x next_y}
                 else
                    eF ;
In
  loop x y } ;
```

Now, the recursive call can begin as soon as we know that eB is True, and all iterations can, in principle, run in parallel.

Barriers are used primarily in M-structure programs for regulating the order in which side-effects are performed (including I/O). Here, insertion or omission of a barrier must be done with care, as it can substantially change the semantics of a program (produce different answers or introduce runtime multiple-put errors).

For programs that do not use M-structures (purely functional, or with I-structures), barriers only change the termination behavior (strictness) of programs. For example:

```
( _ = y ;
  ---
  x = e ; )
```

The barrier causes the expression `e` to become strict in `y`, *i.e.,* `e` does not evaluate until `y` gets a value (from the surrounding context). In extreme cases, of course, insertion of a barrier can introduce deadlock into an otherwise deadlock-free functional or I-structure program.

Barriers are also used in programs that perform explicit storage management (Appendix A.14).

3.8 Sequencing expressions

The `seq` form may be used to sequentialize the evaluation of expressions:

```
{seq e1; e2; ... ; eN}
```

Here, `e1`, `e2`, ..., and `eN` are evaluated sequentially, and the value of `eN` is returned as the value of the whole expression. The values of the other `eJ`'s are discarded. The type of the entire expression is the type of `eN`.

The `seq` form is an abbreviation for:

```
{ _   = e1
  ---
  _   = e2
  ---
        ...
  ---
  foo = eN
In
  foo }
```

4 I-structures

I-structures are a small departure from purely functional semantics. I-structures are layered on top of the purely functional subset of Id by constructs that are distinguished by syntax and by type.

Please refer to Section 3 for general issues concerning non-functional constructs.

4.1 I-structure semantics

A component of a data structure may have *I-structure* semantics (as opposed to *functional* or *M-structure* semantics). For such a component, no value is specified when the data structure is created; instead, a separate assignment statement is used. An I-structure component may be in one of two states: *full* (with a value), or *empty*. All components begin in the empty state (when the data structure is allocated).

An *I-structure* component has a *single assignment* restriction, *i.e.,* it can only be assigned once, at which point its state goes from empty to full. Any attempt to assign it more than once is caught as a runtime error. The component can be read an arbitrary number of times. Further, any attempt to read a component that is empty is automatically blocked until it becomes full. Thus, the programmer does not have to worry about sequencing the reads after the assignment—there is no race condition. I-structure reads and writes are called *I-fetches* and *I-stores*, respectively.

Multiple I-stores into the same location cause a drastic runtime error. The entire program is said to be inconsistent, or $\top$ (see Section 2.38).

4.2 I-structure arrays

I-structure arrays are array-like data structures with empty locations which can be assigned subsequently using I-structure semantics. These arrays are also referred to as I-arrays.

4.2.1 I-array types

An n-dimensional I-array whose components are of type `t` has type:

```
nD_I_array t
```

Synonyms for the type name `1D_I_array`:
`I_vector` `I_array`

Synonym for the type name `2D_I_array`:
`I_matrix`

4.2.2 I-array creation

An n-dimensional I-array is created using the function:

```
nD_I_array::((int,int),
             ...,
             (int,int)) -> (nD_I_array *0)
```

i.e., it takes an index bounds expression (an n-tuple of integer 2-tuples) and returns an empty I-array with those bounds.

It is legal for the index bounds (l, u) along any dimension to be empty, *i.e.,* to have $u = l - 1$ (zero components). However, if $u < l - 1$, the function returns the error value $\perp$ (see Section 2.38).

Synonyms for `1D_I_array` allocator:
`I_vector` `I_array`

Synonym for `2D_I_array` allocator:
`I_matrix`

Example

A 2-dimensional 10×10 I-array:

```
I_matrix ((1,10),(1,10))
```

4.2.3 I-array assignments

Assuming:

```
a     :: (nD_I_array t)
e1    :: (int,...,int)
e2    :: t
```

then the I-array assignment statement:

```
a[e1] = e2 ;
```

uses an I-store operation to assign the value of "`e2`" to the `(j1,...,jn)`'th component of the I-array "`a`", where `(j1,...,jn)` is the value of "`e1`".

If the index `e1` is out of bounds, the statement is equivalent to the error statement $statement_\perp$ (see Section 2.38).

The assignment statement is overloaded for all dimensions of I-arrays and must thus be resolvable by the type checker.

4.2.4 I-array selection

Assuming:

```
a     :: (nD_I_array t)
e1    :: (int,...,int)
```

then the I-array selection expression:

```
a[e1]    :: t
```

uses an I-fetch operation to return the value of the `(j1,...,jn)`'th component of the I-array "`a`", where `(j1,...,jn)` is the value of "`e1`".

If the index `e1` is out of bounds, the selection expression evaluates to the error value $\perp$ (see Section 2.38).

The selection notation is overloaded in two ways— on functional and I-arrays, and on arrays of different dimensions— and must thus be resolvable by the type checker. The corresponding non-overloaded function for I-arrays is:

```
select~nD_I_array ::
    (nD_I_array *0) -> (int,...,int) -> *0
```

4.2.5 I-array index bounds

For each $n \geq 1$, there is a function that returns the index bounds of n-dimensional I-arrays:

```
bounds~1D_I_array ::
    (1D_I_array t) -> (int,int)
bounds~2D_I_array ::
    (2D_I_array t) -> ((int,int),(int,int))
...
```

The overloaded identifier `bounds` represents all these functions.

See Appendix A.9 for standard I-array functions.

Example

An I-array containing j^2 at the j'th index:

```
{  x = I_array (1,10) ;
   {for j <- 1 to 10 do
      x[j] = j * j}
In
   x}
```

4.3 I-structure fields in Algebraic Types

Unlike arrays, where the entire structure has either functional, I-structure or M-structure semantics, an algebraic type can have different fields with different semantics.

4.3.1 Type definition

Recall from Section 2.11 that an algebraic type definition looks like this:

```
type tx tv1 ... tvN = disj1 | ... | disjM;
```

where each disjunct looks like this:

```
tcons t1 ... tL
```

We extend this notation as follows. In each disjunct, each `tJ` may be preceded by "`.`" to indicate that it has I-structure semantics. Such fields are known as I-fields.

4.3.2 Object creation

First, we define *Constructor Terms* as applicative forms:

```
tcons e1 ... eN
```

where `tcons` is a constructor identifier (not an arbitrary expression or identifier) of arity N of some algebraic type.

Objects are created, as usual, by constructor terms. However, an I-field may be left empty by using the special token "_" (underscore). Note: values *must* be supplied for all normal functional components— they cannot be left empty. The typechecker will ensure this.

4.3.3 Component assignment

Assuming:

```
X :: tx tv1 ... tvN
```

and assuming that it is of the form:

```
tcons v1 ... vN
```

then, if its J'th component is an I-field, it may be assigned using an I-store operation using the field assignment statement:

```
X.tcons_J = eV ;
```

The new value in the J'th component of `X` is the value of `eV`.

It is a runtime error if `X` is not a `tcons` disjunct— the statement is equivalent to the error statement $statement_\perp$ (see Section 2.38).

4.3.4 Component selection

Assuming:

```
X :: tx tv1 ... tvN
```

and assuming that it is of the form:

```
tcons v1 ... vN
```

the, if its J'th component is an I-field, it may be selected using an I-fetch operation using the field selection expression:

```
X.tcons_J
```

It is a runtime error if `X` is not a `tcons` disjunct, producing the error value $\perp$ (see Section 2.38).

4.3.5 Component selection in patterns

I-fields may be read using pattern-matching in exactly the same way as functional fields. In other words, an I-field may be matched against a variable (in which case the value is bound to that variable), a constant, another structured pattern, *etc.*

4.3.6 Example: iterative `map`

Here is a tail-recursive implementation of `map list`, using "open lists":

```
type I_list *0 = I_Nil
               | I_Cons *0 .(I_list *0) ;

typeof map~list = (*0->*1)  ->
                  (list *0) -> (list *1) ;

def map~list f Nil   = Nil
 |  map~list f (x:xs)= { iys = I_Cons (f x) _ ;
                          _  = map' f xs iys
                        In
                          I_list_to_list iys } ;

typeof map' = (*0->*1)    ->
              (list *0)   ->
              (I_list *1) -> void ;
```

```
def map' f Nil    L= { L.I_Cons_2 = I_Nil }
 |  map' f (x:xs) L= { L' = I_Cons (f x) _ ;
                       L.I_Cons_2 = L' ;
                       _ = map' f xs L' } ;
```

The function `I_list_to_list` converts an `I_list` type into an ordinarly list type.

4.4 I-structure fields in records

4.4.1 Type definition

A field of a record may be declared to have I-structure semantics by preceding its type with a ".". Such fields are called I-fields.

Example:

```
type foo = {record  fieldf =   int ;
                    fieldi = . int }
```

Here, `foo` is a new record type, containing a functional field `fieldf` and an I-field `fieldi`.

4.4.2 Record creation

In the `record` construction expression, I-fields may be left empty using the special token "_". Example:

```
{record fieldf = 23; fieldi = _ }
```

For convenience, I-fields that are to be left empty may be omitted entirely, so that the above example could be written:

```
{record fieldf = 23}
```

4.4.3 Field assignment

An I-field `fieldi` of a record `X` may be assigned using an I-store operation using the field assignment statement:

```
X.fieldi = eV ;
```

The new value in the field of `X` is the value of `eV`.

4.4.4 Field selection

An I-field `fieldi` of a record `X` may be selected using an I-fetch operation using the expression:

```
X.fieldi
```

(The notation is the same as for normal functional fields).

5 M-structures

M-structures are a major departure from purely functional semantics. M-structures are layered on top of the purely functional subset of Id by constructs that are distinguished by syntax and by type.

Please refer to Section 3 for general issues concerning non-functional constructs.

5.1 M-structure semantics

A component of a data structure may have *M-structure* semantics (as opposed to *functional* or *I-structure* semantics). For such a component, no value is specified when the data structure is created; instead, a separate assignment statement is used. An M-structure component may be in one of two states: *full* (with a value), or *empty*. All components begin in the empty state (when the data structure is allocated).

An M-structure component can be assigned with a *put* operation and read with a *take* operation. A value can be *put* only into an empty component— it is a runtime error if it is already full. Many *take*'s may be attempted concurrently on a component. They all block automatically if the component is empty. When it is full, exactly one of them succeeds in reading the value and the component again becomes empty, so that the other *take*'s remain blocked. It is unspecified as to which *take* amongst competing *take*'s will succeed.

In typical usage of M-structure components, several concurrent computations share a resource. Each computation *takes* it, computes with it, and *puts* it back. The semantics guarantees that each computation has exclusive access to the resource. Note: for correct operation, every *take* must be followed by a *put*, *i.e.,* it is the programmer's responsibility to make sure that these operations are "balanced".

In some situations, the value to be put back is the same as the value taken out (*e.g.,* if we simply want to test the current value in an M-structure field). This combination— taking a value, putting it back, and returning the value— is called an *examine* operation, for which syntactic shorthands are provided.

In some situations, the value to be put back is unrelated to the value taken out (*e.g.,* if we simply want to reset an M-structure field to a known value). This combination— taking out the old value and discarding it, and putting in a new value— is called a *replace* operation, for which syntactic shorthands are provided.

5.2 M-structure arrays

M-structure arrays are array-like data structures with empty locations which can be assigned subsequently using M-structure semantics. These arrays are also referred to as M-arrays.

5.2.1 M-array types

An n-dimensional M-structure array whose components are of type `t` has type:

```
nD_M_array t
```

Synonyms for the type name `1D_M_array`:
    `M_vector`   `M_array`

Synonym for the type name `2D_M_array`:
    `M_matrix`

5.2.2 M-array literals and comprehensions

The array literal notation of Section 2.34.2 is extended to M-arrays by using the keywords `M_vector`, `M_array`, `M_matrix` and `nD_M_array` instead of their functional counterparts:

```
{nD_M_array eBounds of
 eFirst  ...  eLast}
```

The array comprehension notation of Section 2.34.5 is extended to M-arrays by using the keywords `M_array`, `M_vector`, `M_matrix`, `k_M_arrays`, `k_M_vectors`, `k_M_matrices` and `k_nD_M_arrays` instead of their functional counterparts:

```
{k_nD_M_arrays eBounds of
  [e11] = e12 || gen ; ... ; gen
| ...
| [eM1] = eM2 || gen ; ... ; gen }
```

M-array literals and comprehensions are useful for "initializing" M-arrays. The initializations use put operations.

5.2.3 M-array creation

An empty n-dimensional M-array is created using the function:

```
mk_nD_M_array::((int,int),
                 ...,
                 (int,int)) -> (nD_M_array *0)
```

i.e., it takes an index-bounds expression (an n-tuple of integer 2-tuples) and returns an empty M-array with those bounds.

It is legal for the index bounds (l, u) along any dimension to be empty, *i.e.,* to have $u = l - 1$ (zero components). However, if $u < l - 1$, the function returns the error value $\perp$ (see Section 2.38).

Synonyms for `1D_M_array` allocator:
    `mk_M_vector`   `mk_M_array`

Synonym for `2D_M_array` allocator:
    `mk_M_matrix`

Example

A 2-dimensional 10×10 M-array:

```
mk_M_matrix ((1,10),(1,10))
```

5.2.4 M-array assignment

Assuming:

```
a    :: nD_M_array t
e1   :: int,...,int
e2   :: t
```

then the M-array assignment statement:

```
a![e1] = e2 ;
```

uses a put operation to assign the value of "`e2`" to the `j1,...,jn`'th component of the M-array "`a`", where `j1,...,jn` is the value of "`e1`".

An M-array component can be replaced using the statement:

```
a!![e1] = e2 ;
```

which is equivalent to:

```
  i = e1 ;
  v = e2 ;
  ( _ = v
    ---
    _ = a![i] ;
    ---
    a![i] = v )
```

In other words, the old value is taken only after the value of `e2` is available, but note, due to non-strictness, that sub-expressions of `e2` may still be evaluating.

If the index `e1` is out of bounds, the statements are equivalent to the error statement *statement*$_\perp$ (see Section 2.38).

The assignment statement is overloaded for all dimensions of and M-arrays, and must thus be resolvable by the type checker.

5.2.5 M-array selection

Assuming:

```
a      :: nD_M_array t
e1     :: int,...,int
```

then the M-array selection expression:

```
a![e1]     :: t
```

uses a take operation to return the value of the `j1,...,jn`'th component of the M-array "`a`", where `j1,...,jn` is the value of "`e1`".

An M-array component may be examined using the expression:

```
a!![e1]
```

which is equivalent to:

```
{ i      = e1 ;
  v      = a![i] ;
  a![i] = v
In
  v }
```

If the index `e1` is out of bounds, the selection expressions evaluate to the error value $\perp$ (see Section 2.38).

The selection notation is overloaded to work on all dimensions of M-arrays, and must thus be resolvable by the type checker. The corresponding non-overloaded function for M-arrays is:

```
take~nD_M_array ::
    (nD_M_array *0) -> (int,...,int) -> *0
```

5.2.6 M-array index bounds

For each $n \geq 1$, there is a function that returns the index bounds of n-dimensional M-arrays:

```
bounds~1D_M_array ::
    (1D_M_array t) -> (int,int)
bounds~2D_M_array ::
    (2D_M_array t) -> ((int,int),(int,int))
...
```

The overloaded identifier `bounds` represents all these functions.

See Appendix A.10 for standard M-array functions.

5.3 M-structure fields in Algebraic Types

Unlike arrays, where the entire structure has either functional, I-structure or M-structure semantics, an algebraic type can have different fields with different semantics.

5.3.1 Type definition

Recall from Section 2.11 that an algebraic type definition looks like this:

```
type tx tv1 ... tvN = disj1 | ... | disjM;
```

where each disjunct looks like this:

```
tcons t1 ... tL
```

We extend this notation as follows. In each disjunct, each `tJ` may be preceded by "`!`" to indicate that it has M-structure semantics. Such fields are known as M-fields.

5.3.2 Object creation

First, we define *Constructor Terms* as applicative forms:

```
tcons e1 ... eN
```

where `tcons` is a constructor identifier (not an arbitrary expression or identifier) of arity N of some algebraic type.

Objects are created, as usual, by constructor terms. However, an M-field may be left empty by using the special token "_" (underscore). Note: values *must* be supplied for all normal functional components— they cannot be left empty. The type-checker will ensure this.

5.3.3 Component assignment

Assuming:

```
X :: tx tv1 ... tvN
```

and assuming that it is of the form:

```
tcons v1 ... vN
```

then, if its J'th component is an M-field, then it may be assigned using a put operation using the field assignment statement:

```
X!tcons_J = eV ;
```

An M-field can be replaced using the statement:

```
X!!tcons_J = eV ;
```

which is equivalent to:

```
v = eV ;
( _ = v ;
  ---
  _ = X!tcons_J ;
  ---
  X!tcons_J = v )
```

In other words, the old value is taken only after the value of **eV** is available, but note, due to non-strictness, that sub-expressions of **eV** may still be evaluating.

The new value in the J'th component of **X** is the value of **eV**.

It is a runtime error if **X** is not a `tcons` disjunct—the statement is equivalent to the error statement *statement*$_\perp$ (see Section 2.38).

5.3.4 Component selection

Assuming:

```
X :: tx tv1 ... tvN
```

and assuming that it is of the form:

```
tcons v1 ... vN
```

the, if its J'th component is an M-field, then it may be selected using a take operation using the field selection expression:

```
X!tcons_J
```

An M-field may be examined using the expression:

```
X!!tcons_J
```

which is equivalent to:

```
{ v = X!tcons_J ;
  X!tcons_J = v
In
  v }
```

It is a runtime error if **X** is not a `tcons` disjunct, producing the error value $\perp$ (see Section 2.38).

5.3.5 Component selection in patterns

M-fields may be read using pattern-matching, but only a limited form of patterns may be used:

```
_
(!x)
(!!x)
```

where `x` is an *identifier*, not a general pattern. The parentheses in the latter two cases are mandatory.

When the pattern is an "_", it indicates that the M-field is ignored during the pattern matching (no take is performed).

When the pattern is "`(!x)`" or "`(!!x)`", it indicates that the value in the field is to be taken or examined, respectively, and bound to the identifier `x`. However, the take or examine operation is performed only *after* it has been determined that the pattern-matching is successful; the M-field itself plays no role in whether the pattern-matching is successful or not. When a pattern fails to match, *none* of its takes or examines are performed.

5.3.6 Example: unique id generator

A type for data structures for generating unique identifiers that are strings of the form "foo*j*":

```
type uid = Uidcell string !int ;
```

A statement binding **u** to an updatable data structure for generating unique identifiers that are strings of the form "foo*j*":

```
u = Uidcell "foo" _ ;
```

A statement initializing the unique identifier generator **u**:

```
u!Uidcell_2 = 0 ;
```

Generating the next uid foo*j* from the object **u**:

```
{ Uidcell prefix (!j) = u ;
  u!Uidcell_2 = j+1 ;
  uid = conc~string prefix (int_to_string j)
In
  uid }
```

5.3.7 Example: FIFO queue

We can implement a fifo queue using the following type:

```
type queue *0 = Queue  !(open_list *0)
                       !(open_list *0) ;

type open_list *0 = Ocons  .*0
                           .(open_list *0);
```

For example, a fifo queue in which a, b and c have been inserted (in that order) would be:

```
{ tail = Ocons _ _ ;
In
  Queue (Ocons a
             (Ocons b
                    (Ocons c
                           tail)))
        tail }
```

The idea is that the `Queue` structure points at the head and the tail of the queue. The tail is an `Ocons` cell containing an empty slot for the next object to be enqueued, and an empty slot to grow the queue.

Here are the functions to manipulate the queue:

```
def mk_empty_q () = { ol = Ocons _ _
                    In
                      Queue ol ol } ;

def enqueue x q =
    { Queue _ (!tail) = q ;
      newtail = Ocons _ _ ;
      tail.Ocons_1 = x ;
      tail.Ocons_2 = newtail ;
      q!Queue_2 = newtail ;
    In
      () } ;

def dequeue q =
    { Queue (!head) _ = q ;
      Ocons x nexthead = head ;
      q!Queue_1 = nexthead ;
    In
      x } ;
```

Note that the components of the open list structures are each assigned exactly once, so that they have I-structure semantics. However, the two components of the queue structure are repeatedly updated (by every enqueuer and dequeuer), so they have M-structure semantics.

5.4 M-structure fields in records

5.4.1 Type definition

A field of a record may be declared to have M-structure semantics by preceding its type with a "`!`". Such fields are called M-fields.

Example:

```
type foo = {record  fieldf =   int ;
                    fieldm = ! int }
```

Here, `foo` is a new record type, containing a functional field `fieldf` and an M-field `fieldm`.

5.4.2 Record creation

In the `record` construction expression, M-fields may be left empty using the special token "_". Example:

```
{record fieldf = 23; fieldm = _ }
```

For convenience, M-fields that are to be left empty may be omitted entirely, so that the above example could be written:

```
{record fieldf = 23}
```

5.4.3 Field assignment

An M-field `fieldm` of a record `X` may be assigned using a put operation using the field assignment statement:

```
X!fieldm = eV ;
```

An M-field `fieldm` of a record may be replaced using the statement:

```
X!!fieldm = eV ;
```

which is equivalent to:

```
  v = eV ;
  ( _ = v;
    ---
    _ = X!fieldm ;
    ---
    X!fieldm = v ; )
```

In other words, the old value is taken only after the value of `eV` is available, but note, due to non-strictness, that sub-expressions of `eV` may still be evaluating.

The new value in the field of `X` is the value of `eV`.

5.4.4 Field selection

An M-field `fieldm` of a record `X` may be selected using a take operation using the expression:

```
X!fieldi
```

An M-field `fieldm` of a record `X` may be examined using the expression:

```
X!!fieldm
```

which is equivalent to:

```
{ v = X!fieldm ;
  X!fieldm = v ;
In
  v }
```

6 Delayed evaluation

Annotations for delayed evaluation are experimental features of Id to gain experience with infinite structures. Semantically, their only effect is to change the termination behavior of programs. Pragmatically, they can drastically change the runtime resource requirements of a program.

6.1 General Delayed Evaluation

Assuming:

```
e :: t
```

is an expression that evaluates to `v`, then the expression:

```
{# e} :: (delay t)
```

returns `d`, an unevaluated representation of `e` called a *thunk*.

The standard function:

```
force :: (delay *0) -> *0
```

takes a thunk `d`, evaluates the delayed expression in it, and returns `v`, its value. It also "memoizes" the value, so that in multiple evaluations of `(force d)`, the delayed expression itself is evaluated only once. The memoization is transparent—there is no test available to the programmer to determine whether a delayed object has been forced or not.

Note that a delayed object has a different type from the object iteself, and the value is *always* extracted using `force`. Thus, the following two expressions below are incorrect and will be caught by the type-checker:

```
1 + (force 5)  % forcing an undelayed object
1 + {# 5}      % adding a delayed object
```

6.2 Delayed Evaluation For Data Structure Components

While expressions in arbitrary contexts may be delayed using `{# ...}`, it is frequently the case that the delayed object is a component of a data structure. In this situation, one can use a special notation for greater efficiency (less space overhead for thunks) and convenience (no distinction of delayed objects on the basis of type, no need for an explicit `force`).

6.2.1 Delayed components in algebraic types

First, we define *Constructor Terms* as applicative forms:

```
c e1 ... eN
```

where `c` is a constructor identifier (not an arbitrary expression or identifier) of arity N of some algebraic type. Examples:

```
e1 : e2
bnode e1 e2 e3
```

but not:

```
(:) e1          % arity not satisfied
bnode e1 e2     % arity not satisfied
```

(See Section 2.11 for the binary tree algebraic type with constructor `bnode`.)

In a constructor term, any argument may be annotated by "`#`" to indicate that it should be delayed. Example:

```
bnode e1 #e2 e3
```

constructs a binary tree with the left subtree expression delayed.

Unlike general delayed expressions, delayed data-structure components are of the same type as if they were not delayed. There is no explicit `force` operation. A delayed component is evaluated *implicitly* (and stored in the data structure) when an attempt is made to select it (usually in some pattern-match). For example, if `e1::t1`, then

```
bnode  # e1  e2 e3 :: (btree t1)
bnode {# e1} e2 e3 :: (btree (delay t1))
```

In the former, selecting the first component of the `bnode` implicitly evaluates the delayed expression and returns an object of type `t1`, whereas in the latter, it returns an object of type `(delay t1)`, to which `force` must be applied to get an object of type `t1`.

Delayed Lists

List constructor terms may be annotated too:

```
 e1 :  e2    % normal---eager head and tail
 e1 : #e2    % eager head, delayed tail
#e1 :  e2    % delayed head, eager tail
#e1 : #e2    % delayed head and tail
```

For compatibility with the delayed list-comprehension notation (below), the following notations may also be used:

```
e1  :  e2    % normal---eager head and tail
e1  :# e2    % eager head, delayed tail
e1 #:  e2    % delayed head, eager tail
e1 #:# e2    % delayed head and tail
```

Delayed list-comprehensions may be written by changing the opening ":" symbol:

```
{:   e || gen ... }   % normal
{#:  e || gen ... }   % delayed heads
{:#  e || gen ... }   % delayed tails
{#:# e || gen ... }   % delayed heads, tails
```

Assuming:

```
e1::int    eInc::int
```

evaluate to integers $v1$ and $vInc$, respectively, then the expressions:

```
upfrom e1 by eInc       :: (list N)
downfrom e1 by eInc     :: (list N)
```

produce infinite lists containing ($v1$, $v1 + vInc$, $v1 + 2vInc$, ...), and ($v1$, $v1 - vInc$, $v1 - 2vInc$, ...), respectively.

Note: $vInc$ must always be positive.

The short forms:

```
upfrom e1   :: (list N)
downfrom e1 :: (list N)
```

assume that $vInc$ is $+1$.

6.2.2 Delayed components in records

A component of a record may be delayed using "`#`" instead of "`=`":

```
{record
    field1 = e1;
    ...
    fieldJ # eJ;
    ...
    fieldN = eN}
```

Normal and delayed components may be mixed in a single record.

6.2.3 Delayed components in functional arrays

Components of array-comprehensions may be delayed using "`#`" instead of "`=`":

```
{array (l,u) of
  [ei] = ev || gen ...     % normal components
| [ej] # ew || gen ... }   % delayed components
```

Normal and delayed components may be mixed in the same array.

6.2.4 Delayed components in I-structure arrays

The delayed assignment statement (with the same type rules as the ordinary assignment statement, Section 4.2.3):

```
a[e1] # e2
```

stores a thunk for `e2` in the `e1`'th location of `a`. When that location is selected, the thunk is implicitly evaluated, and the value replaces the thunk.

Evaluation on selection

A component of a data structure that has been delayed using any of the above notations is evaluated automatically the first time that it is selected. *This is different from the situation in lazy languages!*. Consider:

```
c1     = e1 #:# e2 ;
(x:_)  = c1
c2     = x  #:# e3 ;
```

Even though all we are doing is copying `e1` from one cons cell to another, it gets evaluated the moment we select it (delaying the head of the `c2` construction is, therefore, pointless).

7 Pragmatics

7.1 Inline substitution

A function definition

```
def foo ... = ... ;
```

may also be written:

```
defsubst foo ... = ... ;
```

in which case the compiler will try to expand the function in place wherever possible. This has no semantic consequence; it merely removes the overhead of function-calls.

The substitution is semantically transparent (it is not a macro). The function itself is still available as a value.

The inlinable functions can be recursive and mutually recursive. The compiler will inline only upto a fixed maximum depth.

7.2 Bounded loops

In principle, all iterations of a loop can run in parallel (subject only to data dependencies). Pragmatically, this may be undesirable as it can swamp the machine. Thus, loops are normally compiled as *bounded loops*, *i.e.,* no more than k iterations may execute simultaneously, for some loop bound k.

Normally, a default loop bound is used. The loop bound can also be specified explicitly using the `bound` keyword. Assuming `ek::int`, then in each of the following forms:

```
{for j <- e bound ek do ... }

{while b bound ek do ... }

{: e || ... ; j <- e bound ek ... }

{array (l,u) of
    ...
  | [ej] = ev || ... ; j <- e bound ek ...
  | ... }
```

the corresponding loop is bounded to k, the value of expression `ek`, which must be a positive integer.

A loop can be forced to execute sequentially using the `sequential` reserved word:

```
{for j <- e sequential do ... }

{while b sequential do ... }

{: e || ...; j <- e sequential ... }

{array ... of
   ...
 | [ej] = ev || ...; j <- e sequential ...
 | ... }
```

Bounded loops are not semantically identical to tail-recursive functions, because they may deadlock where the corresponding tail-recursive function would not. An example using I-arrays:

```
{ A = I_array (0,9) ;
  A[10] = 0 ;
  {for j from 1 to 9 do
     A[j] = f A[j+1] }
In
  A}
```

Another example, in the purely functional subset of Id: Suppose we want to find **biggest**, the maximum of a 100-element array, and **nbig**, the number of elements within 10% of it. Normally, we would traverse the array twice, first computing **biggest**, then using it to compute **nbig**. But non-strictness may tempt us to write a single loop:

```
b,nb = minfloat,0 ;
biggest, nbig =
    {for j <- 1 to 100 do
       next b  = max A[j] b ;
       next nb = if A[j]/biggest < 0.9 then nb
                 else nb+1
     finally b,nb} ;
```

Note that all iterations of the loop use **biggest**, which is itself computed by the loop and is produced by the last iteration. Thus, this loop will deadlock with any k that does not allow it to unfold fully.

We strongly recommend against loops with backward dependencies, *i.e.,* where a variable in an iteration depends on computations in future iterations. However, if an unbounded loop is really necessary, it may be written using the **unbounded** keyword:

```
{for j <- e unbounded do ... }

{while b unbounded do ... }

{: e || ... ; j <- e unbounded ... }

{array (l,u) of
   ...
 | [ej] = ev || ... ; j <- e unbounded ...
 | ... }
```

Unbounded loops are semantically identical to tail-recursive functions.

7.3 Pragmas

A pragma is a statement flagged by "**@**":

```
@identifier
```

or

```
@identifier expression
```

A block of pragmas may be inserted after the formal parameters in a function definition to indicate attributes of the function:

```
def f x1 ... xN {pragma;...;pragma} = e
```

Currently, the only pragma that may be used here is:

```
@inlinable
@inlineable
```

Use of this pragma is equivalent to using the keyword **defsubst** instead of **def**.

7.4 Loop peeling and unrolling

Loop peeling and unrolling are techniques to reduce the overhead of loops. If we think of a loop as equivalent to a definition of a tail-recursive function F and an initial call to F, then loop peeling is similar to inline substitution at the initial call site, and loop unrolling is similar to inline substitution of F in the (recursive) call site inside the body of F. In the limit case, when a loop is completely peeled or unrolled, they are the same.

7.4.1 Loop peeling

The pragma:

```
@peel j
```

when used as a statement in a loop body, may be used to indicate that upto j iterations should be done outside the loop. Here, `j` must be a positive integer constant. For example,

```
{while p do
    @peel 1 ;
    next x = e1 ;
    ...
 finally eFinal}
```

is equivalent to:

```
if p then
    { next_x = e1' ;
      ...
    In
      { x = next_x
      In
        {while p do
             next x = e1 ;
             ...
         Finally eFinal}}}
else
    eFinal
```

where `e1'` uses `next_x` wherever `e1` used `next x`.

When used in `for`-loops with known index bounds, the compiler will usually be able to remove the conditional by optimization.

7.4.2 Loop Unrolling

The pragma:

```
@unroll j
```

when used as a statement in a loop body, may be used to transform the loop so that j iterations of the original loop are performed in a single iteration of the new loop. Here, `j` must be a positive integer constant, or the keyword "`completely`". For example,

```
{while p do
    @unroll 1 ;
    next x = e1 ;
    next y = e2 ;
    ...
 finally eFinal}
```

is equivalent to:

```
{while p do
    next_x' = e1' ;
    next_y' = e2' ;
    ...
    next x,next y =
             { x = next_x' ;
               y = next_y'
             in
               if p then
                   { next_x'' = e1'' ;
                     next_y'' = e2'' ;
                     ...
                   In
                     next_x'',next_y'' }
               else
                   next_x',next_y'}
 finally eFinal}
```

where `e1'` and `e2'` use `next_x'` and `next_y'` whereever `e1` used `next x` and `next y`, respectively (and similarly for `e1''` and `e2''`).

When used in `for`-loops with known index bounds, the compiler will usually be able to remove the conditional by optimization.

The compiler will obey a pragma to unroll the loop completely only if it is a `for` loop with known index bounds.

A Standard Identifiers

Id has standard libraries that implement many useful functions. The names and semantics of these functions are based on the corresponding Common Lisp functions wherever possible. The names for these functions are not reserved words, but for readability and re-usability of code, the programmer is strongly advised not to redefine them.

The compiler should expand these functions *in situ*, so that there is no procedure-calling overhead.

Since the Id libraries are a continuously growing repository of useful functions, the following list is necessarily incomplete. The libraries themselves must be consulted for the current set.

A.1 Booleans

Truth values

```
typeof True = bool;
typeof False = bool;
```

These are also constructors (*i.e.*, they can be used in patterns).

Negation

```
typeof not = bool -> bool;
```

A.2 Numbers

Basic arithmetic

```
typeof negate~int = int -> int;
typeof plus~int   = int -> int -> int;
typeof minus~int  = int -> int -> int;
typeof times~int  = int -> int -> int;
```

```
typeof negate~float = float -> float;
typeof plus~float   = float -> float -> float;
typeof minus~float  = float -> float -> float;
typeof times~float  = float -> float -> float;
```

Comparison

```
typeof eq~int = int -> int -> bool;
typeof ne~int = int -> int -> bool;
typeof lt~int = int -> int -> bool;
typeof le~int = int -> int -> bool;
typeof gt~int = int -> int -> bool;
typeof ge~int = int -> int -> bool;
```

```
typeof eq~float = float -> float -> bool;
typeof ne~float = float -> float -> bool;
typeof lt~float = float -> float -> bool;
typeof le~float = float -> float -> bool;
typeof gt~float = float -> float -> bool;
typeof ge~float = float -> float -> bool;
```

General

```
typeof  pi = float;
typeof 2pi = float;
```

```
typeof odd?  = int -> bool;
typeof even? = int -> bool;
```

```
typeof max~int   = int -> int -> int;
typeof max~float = float -> float -> float;
```

```
typeof min~int   = int -> int -> int;
typeof min~float = float -> float -> float;
```

Implementation-dependent numbers that are the most positive (largest positive numbers):

```
typeof maxint = int;
typeof maxfloat = float;
```

Implementation-dependent numbers that are the most negative (largest negative numbers):

```
typeof minint = int;
typeof minfloat = float;
```

Exponentiation

```
typeof exp = float -> float;
```

where `(exp y)` $\Rightarrow e^y$

Logarithms

```
typeof log   = float -> float;
typeof log10 = float -> float;
```

where `(log x)` $\Rightarrow \log_e x$,
and `(log10 x)` $\Rightarrow \log_{10} x$

Square root, absolute value

```
typeof sqrt      = float -> float;
typeof abs~int   = int   -> int;
typeof abs~float = float -> float;
```

Trignometric functions

Angles are in radians.

```
typeof sin  = float -> float;
typeof cos  = float -> float;
typeof tan  = float -> float;
```

```
typeof asin = float -> float;
typeof acos = float -> float;
typeof atan = float -> float -> float;
```

where (`atan y x`) $\Rightarrow \arctan y/x$, in the range $-\pi$ to $+\pi$. The arguments cannot both be zero.

Hyperbolic functions

```
typeof sinh  = float -> float;
typeof cosh  = float -> float;
typeof tanh  = float -> float;

typeof asinh = float -> float;
typeof acosh = float -> float;
typeof atanh = float -> float -> float;
```

Conversion of integers to floats

```
typeof float = int -> float;
```

Conversion of floats to integers

```
typeof floor = float -> int;
```

where `floor` truncates towards $-\infty$.

```
typeof ceiling = float -> int;
```

where `ceiling` truncates towards $+\infty$.

```
typeof truncate = float -> int;
```

where `truncate` truncates towards 0.

```
typeof round = float -> int;
```

where `round` truncates to the nearest integer, with $x.5$ truncated towards the even integer.

The following four functions are similar to the previous four, except that they return their integer-valued results as floats, for convenience.

```
typeof ffloor = float -> float;
```

where `ffloor` truncates towards $-\infty$.

```
typeof fceiling = float -> float;
```

where `fceiling` truncates towards $+\infty$.

```
typeof ftruncate = float -> float;
```

where `ftruncate` truncates towards 0.

```
typeof fround = float -> float;
```

where `fround` truncates to the nearest integer, with $x.5$ truncated towards the even integer.

Integer division

```
typeof div = int -> int -> int;
```

where (`div x y`) $\Rightarrow q$, where ($q =$ `truncate (x/y)`).

```
typeof quo = int -> int -> int;
```

where (`quo x y`) $\Rightarrow q$, where ($q =$ `floor (x/y)`).

```
typeof rem = int -> int -> int;
```

where (`rem x y`) $\Rightarrow x - qy$, where ($q =$ `truncate (x/y)`).

```
typeof mod = int -> int -> int;
```

where (`mod x y`) $\Rightarrow x - qy$, where ($q =$ `floor (x/y)`).

A.3 Characters

Comparison

```
typeof eq~char = char -> char -> bool;
typeof ne~char = char -> char -> bool;
typeof lt~char = char -> char -> bool;
typeof le~char = char -> char -> bool;
typeof gt~char = char -> char -> bool;
typeof ge~char = char -> char -> bool;
```

The ordering is only guaranteed within the following classes: digit characters, upper case characters, and lower case characters.

Character class predicates

```
typeof digit? = char -> bool;
typeof uc?    = char -> bool;
typeof lc?    = char -> bool;
```

Convert case

```
typeof to_uc~char  = char -> char;
typeof to_lc~char  = char -> char;
```

Character codes

```
typeof char_to_int = char -> int;
typeof int_to_char = int  -> char;
```

A.4 Strings

Comparison

The comparison is lexicographic, based on the ordering of characters.

The following functions are case-sensitive:

```
typeof eq~string = string -> string -> bool;
typeof ne~string = string -> string -> bool;
typeof lt~string = string -> string -> bool;
typeof le~string = string -> string -> bool;
typeof gt~string = string -> string -> bool;
typeof ge~string = string -> string -> bool;
```

The following functions are case-insensitive:

```
typeof eq_ci_string = string -> string -> bool;
typeof ne_ci_string = string -> string -> bool;
typeof lt_ci_string = string -> string -> bool;
typeof le_ci_string = string -> string -> bool;
typeof gt_ci_string = string -> string -> bool;
typeof ge_ci_string = string -> string -> bool;
```

Convert to and from arrays of characters

```
typeof array_to_string =
                (array char) -> string;
```

The argument must have index bounds $(0, n-1)$ when n is the length of the string.

```
typeof string_to_array =
                string -> (array char);
```

The result has index bounds $(0, n-1)$ when n is the length of the string.

Convert to and from lists of characters

```
typeof list_to_string = (list char) -> string;
typeof string_to_list = string -> (list char);
```

Length of string

```
typeof length~string = string -> int;
```

Index string

```
typeof nth~string = int -> string  -> char;
```

First character has index `0`.

Extract substring

Given a starting position and substring length:

```
typeof substring = string ->
                   int    ->
                   int    -> string;
```

Concatenate strings

```
typeof conc~string = string -> string ->
string;
```

Map character function over string

```
typeof map~string = (char->char) ->
                    string       -> string;
```

Convert case

Convert a string to upper or lower case:

```
typeof to_uc~string = string -> string;
typeof to_lc~string = string -> string;
```

Hashing

```
typeof hash~string = int -> string -> int;
```

where `(hash string` n `s)` hashes string `s` into an integer in the range 0 to $(n-1)$.

A.5 Symbols

Comparison

```
typeof eq~symbol = symbol -> symbol -> bool;
typeof ne~symbol = symbol -> symbol -> bool;
```

Hashing

```
typeof hash~symbol = int -> symbol -> int;
```

where `(hash symbol` n `s)` hashes symbol `s` into an integer in the range 0 to $(n-1)$.

Conversion with strings

```
typeof string_to_symbol = string -> symbol;
typeof symbol_to_string = symbol -> string;
```

A.6 Lists

Basic functions

```
typeof Nil    = (list *0);
typeof nil?   = (list *0) -> bool;
typeof cons   = *0 -> (list *0) -> (list *0);
typeof hd     = (list *0) -> *0;
typeof tl     = (list *0) -> (list *0);
```

Length of list

```
typeof length~list = (list *0) -> int;
```

N'th element of list

```
typeof nth~list = int -> (list *0) -> *0;
```

The head is the 0'th element.

First N elements of list

```
typeof first_n = int -> (list *0) -> (list *0);
```

N'th tail of list

```
typeof drop  = int -> (list *0) -> (list *0);
typeof nthtl = int -> (list *0) -> (list *0);
```

`drop n` $=$ `tl`^{n}, so 0'th tail is the list itself.

`nthtl` is a synonym for `drop`.

Last element of list

```
typeof last = (list *0) -> *0;
```

Equality of lists

Given predicate to compare equality of components:

```
typeof eq~list =
 (*0 -> *0 -> bool) ->
 (list *0)          ->
 (list *0)          -> bool;
```

Reverse list

```
typeof reverse = (list *0) -> (list *0);
```

Zipping and unzipping

A family of functions, for each N:

```
typeof zipN =
 (list *1) ->
 ...
 (list *N) -> (list (*1,...,*N));
```

The N input lists should all have the same length. If not, a runtime error occurs— the output list is as long as the shortest input list, and the tail of the last list cell is the error value $\perp$ (see Section 2.38).

```
typeof unzipN =
 (list (*1,...,*N)) -> (list *1,
                         ...    ,
                         list *N);
```

Map function over list

Apply a function to each member of a list, returning list of results in same order:

```
typeof map~list = (*0->*1) ->
                  (list *0) -> (list *1);
```

Filtering

Return only those elements that satisfy predicate:

```
typeof filter =
 (*0 -> bool) ->
 (list *0)    -> (list *0);
```

Return longest prefix of elements that satisfy predicate:

```
typeof takewhile =
 (*0 -> bool) ->
 (list *0)    -> (list *0);
```

Omit longest prefix of elements that satisfy predicate:

```
typeof dropwhile =
 (*0 -> bool) ->
 (list *0)    -> (list *0);
```

Left-associative reduction

```
typeof foldl~list =
 (*0 -> *1 -> *0) ->
 *0               ->
 (list *1)        -> *0;
```

Example:

```
foldl~list f z l
```

returns

```
f (f  (... (f z l0) l1) ...) ln
```

where `l0`, ..., `ln` are the elements of the list `l`.

Right-associative reduction

```
typeof foldr~list =
 (*0 -> *1 -> *1) ->
 *1               ->
 (list *0)        -> *1;
```

Example:

```
foldr~list f z l
```

Returns

```
f l0 (... (f ln z))
```

where `l0`, ..., `ln` are the elements of the list `l`.

Iteration

```
typeof iterate =
 (*0 -> bool) ->
 (*0 -> *0)   ->
 *0           -> (list *0);
```

where `iterate p f x` returns the list containing `x`, `(f x)`, `(f (f x))`, ..., as long as `(p (f`^{n}` x))` is true.

Simultaneous mapping and left-associative reduction

```
typeof map_foldl~list =
 (*0->*1->(*0,*2)) ->
 *0                ->
 (list *1)         -> (*0,list *2);
```

Example:

```
map_foldl~list f z l
```

returns `(zN,m)`, where:

```
z0,m0 = f z  l0
z1,m1 = f z0 l1
  ...
zN,mN = ...
```

`l0`, ..., `lN` are the elements of the list `l`, and `m0`, ..., `mN` are the elements of the list `m`.

For example, if `f` was

```
def f z lj = {  w = z + lj
            In w,w} ;
```

`z` was 0, and `l` contained `1`, `2`, and `3`, then the result `m` would be a list of partial sums: `1`, `3`, and `6`, and the result `zN` would be the sum `6`.

Simultaneous mapping and right-associative reduction

```
typeof map_foldr~list =
 (*1->*0->(*2,*0)) ->
 *0                ->
 (list *1)         -> (list *2,*0);
```

Example:

```
map_foldr~list f z l
```

returns (m,z0), where:

```
m0,z0 = f l0 z1
m1,z1 = f l1 z2
  ...
mN,zN = f lN z
```

l0, ..., lN are the elements of the list l, and m0, ..., mN are the elements of the list m.

For example, if f was

```
def f lj z = {  w = z + lj
             In w,w} ;
```

z was 0, and l contained 1, 2, and 3, then the result m would be a list of partial sums (from back to front): 6, 5, and 3, and the result z0 would be the sum 6.

A.7 Lists as Sets

All these functions require, as their first parameter, an equality function between elements of the sets.

Conversion from list to set

Removes duplicates:

```
typeof settify =
 (*0 -> *0 -> bool) ->
 (list *0)          -> (list *0);
```

Membership test

```
typeof member? =
 (*0 -> *0 ->bool) ->
 *0                ->
 (list *0)         -> B;
```

Union, intersection, difference

```
typeof union =
 (*0 -> *0 -> bool) ->
 (list *0)          ->
 (list *0)          -> (list *0);
typeof intersection =
 (*0 -> *0 -> bool) ->
 (list *0)          ->
 (list *0)          -> (list *0);
typeof difference =
 (*0 -> *0 -> bool) ->
 (list *0)          ->
 (list *0)          -> (list *0);
```

Subset test

```
typeof subset? =
 (*0 -> *0 -> bool) ->
 (list *0)          ->
 (list *0)          -> bool;
```

Set equality test

```
typeof set_equal? =
 (*0 -> *0 -> bool) ->
 (list *0)          ->
 (list *0)          -> bool;
```

A.8 Arrays

In the following, we describe families of functions, for 1D arrays, 2D arrays, *etc.* We describe the entire family using the "nD" meta-syntax. In addition, the substring 1D_array can always be replaced by array or vector, and the substring 2D_array can always be replaced by matrix.

We refer to a sequence of indices for an array by its endpoints "first" and "last", meaning n-tuples containing the lower bounds and upper bounds, respectively, along all dimensions, and stepping the rightmost index fastest.

Index bounds

```
typeof bounds~nD_array =
 (ND_array *0) -> ((int,int),...,(int,int));
```

Synonyms:

```
bounds~1D_array  bounds~vector  bounds~array
bounds~2D_array  bounds~matrix
```

Array component selection

```
typeof select~nD_array =
 (nD_array *0) -> (int,...,int) -> *0;
```

Create k arrays, given filling function

```
typeof make_k_nD_arrays =
 ((int,int),...,(int,int))    ->
 ((int,...,int)->(*1,...,*k)) ->
                          (nD_array *0,
                           ... ,
                           nD_array *k);
```

Example:

```
make_k_nD_arrays b f
```

returns k arrays a1,...,ak with bounds b, such that if

```
f (j1,...,jN) == (v1,...,vk)
```

then

```
ai[j1,...,jN] == vi
```

Synonyms:

	$k = 1$	$k > 1$
$n = 1$	make_array make_vector	make_k_arrays make_k_vectors
$n = 2$	make_matrix	make_k_matrices
$n \geq 1$	make_nD_array	

Equality of arrays

Given predicate to compare equality of elements:

```
typeof eq~nD_array =
 (*0 -> *0 -> bool) ->
 (nD_array *0)      ->
 (nD_array *0)      -> bool;
```

Map function over array

```
typeof map~nD_array =
 (*0 -> *1)    ->
 (nD_array *0) -> (nD_array *1);
```

Example:

```
map~nD_array f a
```

returns an array with same bounds as array `a`, containing `(f a[j])` at each index `j`.

Left-associative reduction

```
typeof foldl~nD_array =
 (*0 -> *1 -> *0) ->
 *0               ->
 (nD_array *1)    -> *0;
```

Example:

```
foldl~nD_array f z a
```

returns:

```
f (f ... (f z a[first]) ... ) a[last]
```

Right-associative reduction

```
typeof foldr~nD_array =
 (*0 -> *1 -> *1) ->
 *1               ->
 (nD_array *0)    -> *1;
```

Example:

```
foldr~nD_array f z a
```

returns:

```
f a[first] ( ... (f a[last] z))
```

Tree-reduction

```
typeof fold~nD_array =
 (*0 -> *1 -> *0) ->
 *0               ->
 (nD_array *1)    -> *0;
```

Example:

```
fold~nD_array f z a
```

reduces the array to a value by first computing the `foldl`s of all the innermost vectors (rightmost index varying), then the `foldl`s of those results with the next innermost index varying, and so on. `fold_...` has more parallelism than `foldl_...` and `foldr_...`.

Simultaneous mapping and left-associative reduction

```
typeof map_foldl~nD_array =
 (*0->*1->(*0,*2)) ->
 *0                ->
 (array *1)        -> (*0,array *2);
```

Example:

```
map_foldl~nD_array f z a
```

returns `(zLast,b)`, where `b` is an array with same bounds as array `a`, and

```
zFirst, b[first]  = f z           a[first]
zSecond,b[second] = f zFirst      a[second]
...
zLast,  b[last]   = f zLastButOne a[last]
```

For example, if `f` was

```
def f z aj = {  w = z + aj
             In w,w} ;
```

`z` was 0, and `a` was a vector containing `1`, `2` and `3`, then the result `b` would be a vector of partial sums: `1`, `3` and `6`, and the result `zLast` would be the sum `6`.

Simultaneous mapping and right-associative reduction

```
typeof map_foldr~nD_array =
 (*1->*0->(*2,*0)) ->
 *0                ->
 (array *1)        -> (array *2,*0);
```

Example:

```
map_foldr~nD_array f z a
```

returns `(b,zFirst)`, where `b` is an array with same bounds as array `a`, and

```
b[first], zFirst   = f a[first]  zSecond
b[second],zSecond  = f a[second] zThird
...
b[last],  zLast    = f a[last]   z
```

For example, if f was

```
def f z aj = {  w = z + aj
             In w,w} ;
```

z was 0, and a was a vector containing 1, 2 and 3, then the result b would be a vector of partial sums (from last to first): 6, 5 and 3, and the result zFirst would be the sum 6.

A.9 I-structure arrays

I-array allocators

```
typeof nD_I_array =
 ((int,int),...,(int,int)) -> (nD_I_array *0);
```

Synonyms for 1D_I_array:
`I_vector` `I_array`

Synonym for 2D_I_array:
`I_matrix`

I-array index bounds

```
typeof bounds~nD_I_array =
 (nD_I_array *0) -> ((int,int),...,(int,int));
```

I-array component selection

```
typeof select~nD_I_array =
 (nD_I_array *0) -> (int,...,int) -> *0;
```

Fill rectangular region of *k* I-arrays given filling function

```
typeof fill~k_nD_I_arrays =
 ((int,int),...,(int,int))            ->
 ((int,...,int) -> (*1,...,*k))       ->
 (nD_I_array *0, ... , nD_I_array *k) -> void;
```

Example:

```
fill~k_nD_I_arrays r f (a1,...,ak)
```

fills region r of I-arrays a1,...,ak such that if

```
f (j1,...,jN) == (v1,...,vk)
```

then

```
ai[j1,...,jN] == vi
```

A.10 M-structure arrays

M-array allocators

```
typeof nD_M_array =
 ((int,int),...,(int,int)) -> (nD_M_array *0);
```

Synonyms for 1D_M_array:
`M_vector` `M_array`

Synonym for 2D_M_array:
`M_matrix`

M-array index bounds

```
typeof bounds~nD_M_array =
 (nD_M_array *0) -> ((int,int),...,(int,int));
```

M-array component selection

```
typeof take~nD_M_array =
 (nD_M_array *0) -> (int,...,int) -> *0;
```

A.11 Object identity

```
typeof same? = *0 -> *0 -> bool;
```

A.12 Delayed Evaluation

```
typeof force = (delay *0) -> *0;
```

A.13 Input/Output

The following is a temporary library of input/output types and procedures. This will be replaced by a new I/O library after we have more experience with this one.

For synchronization of concurrent I/O operations, every procedure takes an extra "trigger" argument and returns an extra "signal" result, both of type f_status. The procedures are all strict in the trigger argument, *i.e.*, they do not attempt any I/O until it is available. Further, the procedures do not return the signal result until they have "done" the I/O. Thus, an application can perform I/O operations in a particular order by threading an f_status token through all of them. An application can perform I/O operations in a non-deterministic order by choosing *not* to thread this argument through them.

In each of the procedures, the result **f_status** indicates how the i/o operation went (ok, end-of-file, or error).

I/O types

There is a built-in type called **fstream**.

The following type is used to test whether an I/O operation succeeded or not.

```
type f_status = F_ok | F_err | F_end ;
```

Opening I/O streams

Access modes for opening files:

```
type access_mode = For_Read
                 | For_Write
                 | For_Append ;
```

Opening an I/O stream on a file:

```
typeof fopen = string ->
               access_mode ->
               (fstream,f_status) ;
```

The string argument is the file name.

Opening an input stream on a string:

```
typeof sopen_in = string ->
                  (fstream, f_status) ;
```

The string argument is treated as the "input file" for the stream.

Opening an output stream to a string:

```
typeof sopen_out = int ->
                   (fstream,f_status,string) ;
```

The integer argument specifies the desired length of the string. The string is returned as the third component of the result tuple, while the stream that writes into the string is returned as the third component. Because of non-strictness, the string is returned immediately, but is empty. As output operations on the stream are performed, the string is filled up.

Closing I/O streams

```
typeof fclose = fstream ->
                f_status ->
                f_status ;
```

The trigger argument is used for sequentialization. By threading the **f_status** result of the last i/o operation into the trigger of **fclose**, the programmer can ensure that the i/o operations on the stream have been completed before it is closed. If the stream was an output stream into a string, the remainder of the string is padded with null characters.

Positioning

When an I/O stream is opened, it is initially positioned at the first character, *i.e.,* the next character to be read is the first one. The position may be moved in front of the j'th character using the function:

```
typeof fseek = fstream ->
               int ->
               f_status ->
               f_status ;
```

The desired position j is supplied as the integer argument. A status of **F_eof** is returned if j is beyond the end of the stream.

The current position of an i/o stream may be queried using the function:

```
typeof fposition = fstream ->
                   f_status ->
                   (int,f_status) ;
```

Input

The following functions read (parse) a character, an integer, a float and a string, respectively, from a stream of characters:

```
typeof fscan_char   = fstream ->
                      f_status ->
                      (char,f_status) ;
typeof fscan_int    = fstream ->
                      f_status ->
                      (int,f_status) ;
typeof fscan_float  = fstream ->
                      f_status ->
                      (float,f_status) ;
typeof fscan_string = fstream ->
                      f_status ->
                      (string,f_status) ;
```

In each case, the first component of the result 2-tuple is the value that was read, and is meaningful only if the result **f_status** is **F_ok**.

For **fscan_char**, a status of **F_end** is returned if no characters remain in the stream.

For **fscan_string**, a status of **F_end** is returned if no characters remain in the stream. If any characters remain, the input is consumed upto and including the next newline, if any, or the end of the stream, otherwise. The consumed characters, minus the trailing newline, are returned in the result string.

For **fscan_int**, leading whitespace characters are skipped (spaces, tabs, newlines). Then, a status of **F_end** is returned if no characters remain in the stream. Otherwise, it scans an integer:

[+/-] [whitespace] digit+

For **fscan_float**, leading whitespace characters are skipped (spaces, tabs, newlines). Then, a status of **F_end** is returned if no characters remain in the stream. Otherwise, it scans a float:

[+/-] [whitespace] digit+ [. digit*]

The following function reads the next character without consuming it:

```
typeof fpeek_char  = fstream ->
                     f_status ->
                     (char,f_status) ;
```

Output

The following functions print a character, an integer, a float and a string, respectively, into a stream of characters:

```
typeof fprint_char   = fstream ->
                       char ->
                       f_status ->
                       f_status ;
typeof fprint_int    = fstream ->
                       int ->
                       f_status ->
                       f_status ;
typeof fprint_float  = fstream ->
                       float ->
                       f_status ->
                       f_status ;
typeof fprint_string = fstream ->
                       string ->
                       f_status ->
                       f_status ;
```

For **fprint_int**, a leading sign is printed only if the number is negative.

For **fprint_float**, a leading sign is printed only if the number is negative. The number of digits of mantissa printed is unspecified.

The following function prints out a newline:

```
typeof fprint_nl  = fstream ->
                    f_status ->
                    f_status ;
```

Formatted I/O

Formatted input involves parsing numbers and strings from an input stream of characters, and formatted output involves writing numbers and strings to an output stream of characters, according to a list of format directives.

Items to be read or written are placed in a list `PRINT_ITEM` objects:

```
type PRINT_ITEM = Pri int
                | Prf float
                | Prs string
                | Prc char
                | Prnl ;
```

These are for integers, floats, strings, characters and newlines, respectively. A possible extension in the future is to have additional disjuncts with field width, centering and padding specifications.

Formatted output

```
typeof fprintf = fstream ->
                 (list PRINT_ITEM) ->
                 f_status ->
                 f_status ;
```

Example: Suppose `i` and `x` evaluate to 23 and 6.847, respectively. The statement:

```
stat = fprintf ( Prs "i = "
               : Pri i
               : Prs ", x = "
               : Prf x
               : Prnl
               : Prs "Voila!"
               : Prnl
               : Nil )
               trig ;
```

produces output that looks like this:

```
i = 23, x = 6.847
Voila!
```

Formatted input

The desired scanning (parsing) of the input is specified in a list of `SCAN_ITEM` objects:

```
type SCAN_ITEM = Sci
               | Scf
               | Scs
               | Scc
```

These are for integers, floats, strings and characters, respectively. A possible extension in the future is to have additional disjuncts with field width, matching, termination and skipping specifications.

Formatted input is performed by the following function:

```
typeof fscanf = fstream ->
                (list SCAN_ITEM) ->
                f_status ->
                (list PRINT_ITEM, f_status) ;
```

The input is scanned according to the `SCAN_ITEM` list, and the results are returned in a `PRINT_ITEM` list.

Example: the statement:

```
items, stat = fscanf file1 (Sci:Scf:Nil) trig ;
```

scans an integer and a floating point number from the stream, and returns a list of two `PRINT_ITEM`s, the components of which can be accessed by the pattern-binding statement:

```
(Pri j:Prf x:Nil) = items ;
```

This binds the numbers that were read to `j` and `x`, respectively.

Standard input and output

There are three standard streams, usually connected to the terminal:

```
typeof stdin  = fstream;
typeof stdout = fstream;
typeof stderr = fstream;
```

For each of the functions for I/O to streams, there is a corresponding function for I/O to `stdin` and `stdout`, respectively, by omitting the leading "`f`" in the function name and omitting the `fstream` argument:

```
typeof  scan_char   = f_status ->
                      (char,f_status) ;
typeof  scan_int    = f_status ->
                      (int,f_status) ;
typeof  scan_float  = f_status ->
                      (float,f_status) ;
typeof  scan_string = f_status ->
                      (string,f_status) ;

typeof  print_char   = char ->
                       f_status ->
                       f_status ;
typeof  print_int    = int ->
                       f_status ->
                       f_status ;
typeof  print_float  = float ->
                       f_status ->
                       f_status ;
typeof  print_string = string ->
                       f_status ->
                       f_status ;
typeof  print_nl     = f_status ->
                       f_status ;

typeof scanf  = (list SCAN_ITEM) ->
                f_status ->
                (list PRINT_ITEM,f_status) ;
typeof printf = (list PRINT_ITEM) ->
                f_status ->
                f_status ;
```

A.14 Storage Management

The following storage management procedures are very dangerous— they are calls to storage manager that permit deallocation and reuse of heap objects instead of relying on general garbage collection. If the programmer is not careful, programs that use these procedures can fail in bizarre ways.

Return a heap object to the storage allocator:

```
typeof dealloc = *0 -> void ;
```

If the argument is not a heap object (*e.g.,* it is a number), no action is performed. Note: this only deallocates the object corresponding directly to the argument; it does not transitively deallocate any other objects to which this object may point. The `void` result is returned only after the deallocation has been completed.

Clear a heap object so that all its slots are once again empty:

```
typeof clear = *0 -> *0 ;
```

If the argument is not a heap object (*e.g.,* it is a number), no action is performed. Note: this only clears the object corresponding directly to the argument; it does not transitively clear any other objects to which this object may point. As a result of this clearing, of course, all previous contents of this object are lost. The result is a pointer to the same object, and is returned only after the clearing has been completed.

Copy a heap object completely:

```
typeof copy = *0 -> *0 ;
```

If the argument is not a heap object (*e.g.,* it is a number), no action is performed. Note: this only copies the object corresponding directly to the argument; it does not transitively copy any other objects to which this object may point. The result is a pointer to the new object, and is returned as soon as it is allocated. The procedure reads the old object using I-fetches.

B List of overloaded operators and identifiers

In almost all cases, the types of the instances of an overloaded operator or identifier have some common structure. Thus, to abbreviate the listings below, for each overloaded operator or identifier, we show a general type scheme using meta-variable T, and then we list all the instantiations of T.

For example, the entry:

```
max:
typeof max~T = T -> T -> T;
```

where T = `int float`

indicates that the overloaded identifier `max` stands for the two non-overloaded identifiers:

```
typeof max~int   = int   -> int   -> int;
typeof max~float = float -> float -> float;
```

B.1 Overloaded operators

Unary prefix -:

```
typeof negate~T = T -> T;
```

where T = `int float`

Binary infix +, - and *:

```
typeof plus~T  = T -> T -> T;
typeof minus~T = T -> T -> T;
typeof times~T = T -> T -> T;
```

where T = `int float`

Binary infix == and <>:

```
typeof eq~T = T -> T -> bool;
typeof ne~T = T -> T -> bool;
```

where T = `int float char bool string symbol`

Binary infix <, <=, > and >=:

```
typeof lt~T = T -> T -> bool;
typeof le~T = T -> T -> bool;
typeof gt~T = T -> T -> bool;
typeof ge~T = T -> T -> bool;
```

where T = `int float char string`

B.2 Overloaded identifiers

Overloaded identifiers are listed below in alphabetic order.

```
abs:
```

```
typeof abs~T = T -> T;
```

where T = `int float`

`bounds:`

```
typeof bounds~T = (T *0) -> (int,int);
```

where T = `1D_array 1D_I_array 1D_M_array`

```
typeof bounds~T =
  (T *0) -> ((int,int),...,(int,int));
```

where T = `nD_array nD_I_array nD_M_array`
for all $n > 1$

`conc:`

```
typeof conc~T = T -> T -> T;
```

where T = `string (list *0)`

`eq:`

```
typeof eq~T = T -> T -> bool;
```

where T = `int float char bool string symbol`

```
typeof eq~T =
  (*0 -> *0 -> bool) ->
  (T *0)             ->
  (T *0)             -> bool;
```

where T = `list nD_array`

`fill:`

```
typeof fill~k_nD_arrays =
 ((int,int),...,(int,int))          ->
 ((int,...,int) -> (*1,...,*k))      ->
 (nD_I_array *0, ... , nD_I_array *k) -> void;
```

where T = `nD_I_array`
for all $k \geq 1$, $n \geq 1$

`foldl:`

```
typeof foldl~T =
  (*0 -> *1 -> *0) ->
  *0               ->
  (T *1)           -> *0;
```

where T = `list nD_array`
for all $n \geq 1$

`foldr:`

```
typeof foldr~T =
  (*0 -> *1 -> *1) ->
  *1               ->
  (T *0)           -> *1;
```

where T = `list nD_array`
for all $n \geq 1$

`hash:`

```
typeof hash~T = int -> T -> int;
```

where T = `string symbol`

`length:`

```
typeof length~T = T -> int;
```

where T = `string list`

`map:`

```
typeof map~string =
  (char -> char) -> string -> string;
typeof map~list =
  (*0->*1) -> (list *0) -> (list *1);
typeof map~nD_array =
  (*0->*1) -> (nD_array *0) -> (nD_array *1);
```

for all $n \geq 1$

`map_foldl:`

```
typeof map_foldl~T =
  (*0->*1-> (*0,*2)) ->
  *0                 ->
  (T *1)            -> (*0,list *2);
```

where T = `list nD_array`
for all $n \geq 1$

`map_foldr:`

```
typeof map_foldr~T =
  (*1->*0->(*2,*0)) ->
  *0                ->
  (T *1)          -> (list *2, *0);
```

where T = `list nD_array`
for all $n \geq 1$

`max:`

```
typeof max~T = T -> T -> T;
```

where T = `int float`

`min:`

```
typeof min~T = T -> T -> T;
```

where T = `int float`

`nth:`

```
typeof nth~string = int -> string -> char;
typeof nth~list   = int -> (list *0) -> *0;
```

`select:`

```
typeof select~T = (T *0) -> int -> *0;
```

where T = `nD_array nD_I_array`
for all $n \geq 1$

`to_uc:`

```
typeof to_uc~T = T -> T;
```

where T = `char string`

`to_lc`

```
typeof to_lc~T = T -> T;
```

where T = `char string`

B.3 Overloaded array notations

Functional and I-structure array selection expressions:

```
A[i]
```

are overloaded to work on:

```
(nD_array *0)
(nD_I_array *0)
```

for $n = 1, 2, \ldots$

M-structure array selection expressions:

```
A![i]
A!![i]
```

are overloaded to work on:

```
(nD_M_array *0)
```

for $n = 1, 2, \ldots$

I-structure array assignment statements:

```
A[i] = v;
```

are overloaded to work on:

```
(nD_I_array *0)
```

for $n = 1, 2, \ldots$

M-structure array assignment statements:

```
A![i] = v;
A!![i] = v;
```

are overloaded to work on:

```
(nD_M_array *0)
```

for $n = 1, 2, \ldots$

C Incompatible Changes

C.1 Changes from Id 90.0 to Id 90.1

Reserved words

The following are now reserved words:

```
instance
instances
M_array
M_matrix
M_vector
record
sequential
```

The following families are now reserved words:

*k*_*n*`D_M_arrays`
k`_M_vectors`
k`_M_arrays`
k`_M_matrices`
n`D_M_array`

The following are no longer reserved words:

```
assign
error
put
```

Char, String, Symbol notations

We have abandoned Common Lisp notation and switched completely to ANSI C notation for character and string constants. The use of single quotes for character constants also necessitated a change in the notation for symbols.

Failing patterns in comprehensions

When a pattern fails in a generator in a list or array comprehension, a runtime error occurs. Previously, this was not specified. Also, this differs from the convention in some other functional languages where failing patterns are silently dropped (*i.e.*, failing patterns are used as filters).

M-array allocator names

The M-array allocators have been renamed to:

```
mk_m_array
mk_m_vector
mk_m_matrix
mk_nD_m_array
```

because of a conflict with the reserved words:

```
m_array
m_vector
m_matrix
nD_m_array
```

I-structure and M-structure field declaration

In algebraic types, I-structure and M-structure fields are signalled by "`.`" and "`!`", respectively. Previously, they were signalled by "`!`" and "`!!`", respectively.

M-structure assignment and selection notation

M-structure assignment and selection now use "`!`" explicitly, *e.g.*,

```
A![j] = A![j] + 1;
B!age = B!age + 1;
```

whereas previously they had the same notation as for functional data structures and I-structures.

Storage management pragmas

The following storage management pragmas have been removed:

```
@release
@circulate
```

The standard procedures in Appendix A.14 take their place. These are used in conjunction with barriers to achieve the same effect.

C.2 Changes from Id 88.x to Id 90.0

Overloading

There is now a clear position on overloading, and it is not as ambitious as originally expected. Only built-in operators and identifiers are overloaded. The type-checker must be able to resolve overloading locally. If it cannot, the programmer must assist the type checker with explicit type declarations or by using a non-overloaded identifier instead.

Numeric types

The single numeric type `N` has been replaced with two separate types **`int`** and **`float`**. The Id 88.x manual had indicated that this was likely to happen.

Various operators and identifiers are now overloaded to work on both integers and floats.

Identifiers

The identifier "`?`" consisting of only a question mark is no longer treated specially.

Character and String constants

Character and string constants now follow the convention in the C programming language instead of Common Lisp.

Escape sequences are available in character and string constants, following the convention in the C programming language.

Symbols

Symbols now begin with a backslash, instead of a single quote.

Lists

The lexical token "`!`" is no longer an operator. The standard function **`nth~list`** is now available instead for indexing lists.

Equality and inequality

The operators "==" and "<>" are now overloaded only on a few primitive types (they no longer work on lists, arrays, tuples or other algebraic types). The overloaded identifier **`eq`** works for the primitive types as well as for lists and arrays. The standard identifiers **`eq~list`** and **`eq~array`** are available for list and array equality. The programmer must write equality functions for other types.

Patterns

Floating point constants are no longer allowed in patterns.

The don't care pattern "_" consisting of a single underscore has taken on additional meaning. For M-structure components of an algebraic type, it also means that no *take* operation is to be performed on that component.

The lexical token "_"

In patterns, the "don't care" pattern "_" now has additional meaning. For M-structure components of an algebraic type, it also means that no *take* operation is to be performed on that component.

In expressions, "_" no longer stands for the void value (use "()" instead). In constructor terms for algebraic types, it is used to indicate that no value is to be stored into the corresponding I-structure or M-structure component of the object.

Void

The type `void` now has a distinguished constant "()" representing the only value in that type. It can be used in patterns. The previous "_" notation for void values is no longer available.

`Call` statements

The statement form:

```
call e
```

is no longer available. Use this instead:

```
_ = e
```

Standard identifiers

In accordance with our new convention for systematically relating overloaded identifiers to the corresponding non-overloaded ones, the names of certain standard functions have changed. Examples:

New	*Old*
`bounds~nD_array`	`nD_bounds`
`bounds~nD_I_array`	`nD_bounds`
`map~list`	`map_list`

Comprehension notation

We now use semicolons (`;`) instead of ampersands (`&`) to separate generators in list and array comprehensions. Ampersand is no longer a lexeme.

In array comprehensions and accumulator comprehensions, bounds expressions are now followed by the keyword "`of`", bringing it more in line with the notation for `case` expressions.

In accumulators, in accumulation clauses, we use the notation:

```
[eis] = evs || gen ; ... ; gen
```

instead of

```
eis gets evs || gen ; ... ; gen
```

Accordingly, `gets` is no longer a reserved word.

C.3 Changes from Id Noveau to Id 88.x

Function definitions (see Section 2.31) are now *always* introduced by the `def` keyword. Previously, function definitions at the top-level had the keyword, whereas function definitions inside blocks did not.

The terms `array`, `vector`, `matrix`, `k_nD_arrays`, *etc.* are now keywords introducing array comprehensions (see Section 2.34.5).

We are now serious about type-checking. Unless you explicitly disable it, every program must now pass the type-checker. Some existing programs will now be rejected by the type-checker. Typically, such programs violate the restriction that all components of a list and all components of an array must be homogeneously typed. This restriction was mentioned in the Id Nouveau document, but was not enforced by the compiler.

For-loop syntax has changed (generalized). Instead of:

```
for j from e1 to e2 by eInc do
```

we now say:

```
for j <- e do
```

where `e` is *any* list expression. The phrase:

```
e1 to e2 by eInc
```

is now a full-fledged expression (*i.e.,* it can be used anywhere), and denotes a list containing the arithmetic series from `e1` through `e2` with `eInc` increment.

References

[1] Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, CA, December 1978.

[2] R. S. Nikhil. Id Nouveau Reference Manual, Part I: Syntax. Technical report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[3] R. S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[4] R. S. Nikhil and Arvind. Id/83s. Technical report, MIT Laboratory for Computer Science, Cambridge, MA 02139, July 1985. (Prepared for MIT Subject 6.83s).

[5] R. S. Nikhil and Arvind. *Programming in Id: a parallel programming language*. 1990. (book in preparation).

[6] R. S. Nikhil, K. K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, MIT Laboratory for Computer Science, Cambridge, MA 02139, July 23 1986. (Prepared for MIT Subject 6.83s).