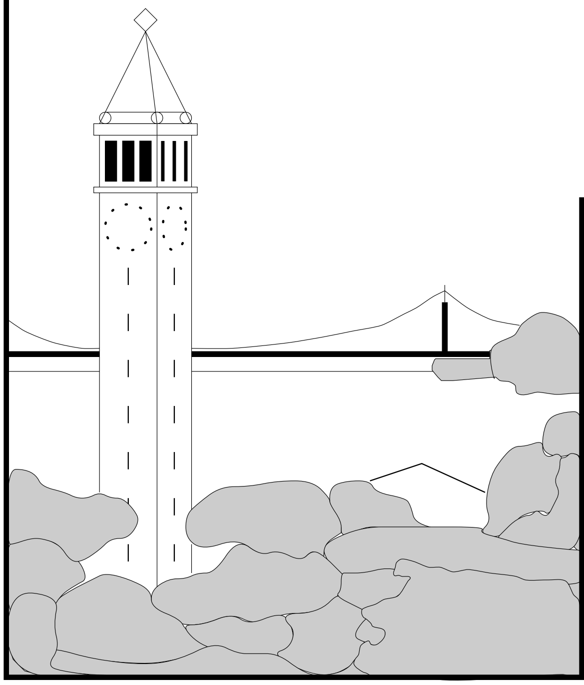
FIDIL Reference Manual

Paul N. Hilfinger   *Phillip Colella*



Report No. UCB/CSD 93-759

2 May 1993

Computer Science Division (EECS)
University of California
Berkeley, California 94720

FIDIL Reference Manual [1]

Paul N. Hilfinger Phillip Colella

2 May 1993

[1]This work funded by NSF (DARPA) grant DMS-8919074.

Contents

Chapter 1

Introduction

FIDIL (for FInite DIfference Language) is a language supporting finite difference and particle method computations. It extends the semantic domain of FORTRAN-like algebraic languages with facilities for construction, composition, refinement, and other manipulation of grids—called *domains*—and for performing computations on functions defined over these domains. FIDIL is an attempt to automate much of the routine bookkeeping that forms a large part of many programs involving PDEs, and to bring the semantic level of these programs closer to that at which the algorithms are conceived and published.

This report gives the current definition of the FIDIL language. We expect the definition to evolve rapidly with experience.

Chapter 2

Notation

In BNF syntactic descriptions, a construct of the form “$\{C \ldots\}$” indicates 0 or more instances of the C. A construct of the form “$\{C\ \sigma \ldots\}$,” where σ is a punctuation mark, indicates 0 or more instances of C separated by σ characters. A superscripted ‘+’ after the closing brace indicates 1 or more instead of 0 or more. A construct surrounded by square brackets ([ ]) indicates 0 or 1 instances of the construct. A list of BNF clauses separated by vertical bars (|) denote alternatives. Set braces, square brackets, vertical bars, and other meta-syntactic marks that are intended as terminal symbols in the grammar are surrounded by single quotes (e.g., ‘[’ and ‘]’). Other terminal symbols may also be placed in single quotes, as clarity dictates.

Non-terminal symbols appear within angle brackets (<>). Parts of the symbol that appear in slanted type are comments for syntactic purposes, but may have semantic significance. For example, “<*infix* operator>” is syntactically any operator, but must also appear within the scope of a declaration for that operator as an infix operator. Certain non-terminals are not defined; their definitions (which may be context-sensitive) are implied by their names.

Chapter 3

Lexical Details

FIDIL source text is generally free-form. Blanks or punctuation must delimit each end of all keywords, identifiers, and numbers, and may not appear within them. Blanks or non-operator characters (see below) must delimit each end of all operators, and may not appear within them. Beginnings and ends of lines, comments, and tabs all count as blanks outside of string literals. Blanks are insignificant except as delimiters or characters in string literals. Keywords, which in the rest of this document are indicated by bold face type, are reserved.

Identifiers have the following, fairly conventional, syntax.

<identifier> ::= <letter> { <letter or digit> ...}
<letter> ::=
    'a' | ... | 'z' | 'A' | ... | 'Z' | '_'
<digit> ::=
    '0' | ... | '9'

Identifiers may be of any length. The cases of letters composing the identifier are significant; for example the identifier 'x' is distinct from 'X'.

Numeric literals are formed according to the following syntax.

<number> ::=
    <integer literal> ['i']
  | <real literal> ['i']
<integer literal> ::=
    { <digit> ...}$^+$

<real literal> ::=
<integer literal> <exponent>
| <integer literal> '`.`' [<integer literal> [<exponent>]]
| '`.`' <integer literal> [<exponent>]
<sign> ::= '`+`' | '`-`'
<exponent> ::=
`E` [[<sign>] <integer literal>]
| `D` [[<sign>] <integer literal>]

The letter 'i' appended to a number (without intervening blanks) indicates a purely imaginary quantity. The letter 'D' in the exponent of a real literal indicates a long real quantity and 'E' denotes a short real quantity. In the absence of such an indication, 'D' is assumed. All of the letters 'i', 'E', and 'D' in these contexts may appear in either case. When the integer literal in an exponent is missing, it defaults to 0. Programmers can specify signed numbers as expressions involving a numeric literal and a unary minus or plus operator.

String literals denote arrays of characters (type `[1 ..` n`]` **char**, for $n \geq 0$ the number of characters.) They have the following syntax.

<string literal> ::=
'`"`' { <string literal character> ...} '`"`'
<string literal character> ::=
<any character other than `"` and end of line>
| `'\'` <any character other than end of line>

Characters following a backslash are interpreted as in C.

Comments begin with '`/*`' and end with '`*/`'. They may span any number of lines. A '`/*`' sequence inside a comment is ignored.

Chapter 4

Basic Program Structure

Syntax

<compilation> ::= { <compilation item> ';' ...} [';']
<compilation item> ::=
    <outer declaration>
  | **export** { <identifier> ',' ...}$^+$
<outer declaration> ::=
    [**external**] <variable declaration>
  | <outer constant declaration list>
  | <type declaration>
  | <operator declaration>

Semantics

A *compilation* is a collection of declarations of variables, subprograms, and other constants, together with directives for linking together declarations in separately processed compilations. A *program* is a collection of one or more compilations that is sufficiently complete to execute.

Each outer declaration has a scope that begins at the defining instance of the entity declared and continues to the end of the compilation. The **exports** clause extends the scope of the declaration of any identifier listed to other compilations that reference the identifier in *external declarations.* Types may not be exported (but the `#include` preprocessor directive can give the effect of exporting and importing types.) An external declaration is either a variable declaration prefixed by the keyword **external** or a subprogram constant defined with a literal whose body is external (see §8.2). In any compilation, there must be exactly one non-external declaration of each identifier in an **exports** list (such a declaration is called an *exported declaration*).

When the compilations forming a program are linked, there must be at most one exported declaration for any identifier. There must be an exported declaration for every external variable and for every external subprogram for which there is a call in one of the compilations. The type and class (variable or constant) of every external declaration must match that of the corresponding exported declaration.

Every program (but not every compilation) must contain a distinguished procedure *main*. Execution of the program consists of first executing all outer declarations in order (that is, in textual order within each compilation, with compilations executed in an order specified to the linker) and then calling the procedure *main*.

Examples

```
export current_parameters, force_fn, nd;

let
    force_fn = proc (Position x) -> Force: ···;
    /* Exported function. */

[1 .. 3] long real current_parameters;
    /* Exported variable */
```

Chapter 5

Preprocessing

It is often useful to be able to group shared definitions in *header files* that can be incorporated in other source files without fear of transcription error. For this purpose, FIDIL uses the same preprocessing as is provided by the C language. Source lines beginning with the character '#' are assumed to be preprocessor directives. A directive of the form

`#include "`*file-name*`"`

will insert the contents of the named file in place of the directive.

Chapter 6

Declarations

Syntax

<outer constant declaration list> ::=
    let { <outer constant declaration> ',' ...$\}^+$
<constant declaration list> ::=
    let { <constant declaration> ',' ...$\}^+$
<constant declaration> ::=
    <identifier> = <expression>
  | '(' { <identifier> ',' ...$\}^+$ ')' = <expression>
  | <operator> = <*subprogram* expression>
<outer constant declaration> ::=
    <constant declaration>
  | <identifier> = <generic subprogram literal>
  | <operator> = <generic subprogram literal>
<variable declaration> ::=
    <type> { <identifier> ',' ...$\}^+$
<type declaration> ::=
    type { <identifier> = <type> , ...$\}^+$

Semantics

A *scope* is a section of program text. Certain program constructs determine a *defining scope*. For example, the text of a subprogram literal is a defining scope. The *scope of a declaration* is the section of program text to which it applies. It begins at the point of the declaration (i.e., it includes the text of the declaration itself) and continues to the end of the innermost defining scope in which that declaration appears. The meaning of a given instance of an identifier or operator is governed by the declarations in whose scope it appears. When more than one declaration is present, the one that applies is determined by the rules for resolving ambiguity in §6.1.

Some declarations do not themselves indicate the creation of a declared entity, but rather make reference to another declaration that appears elsewhere for the purpose of enlarging that declaration's scope. We refer to such definitions collectively as *incomplete* declarations, and all others as *complete* declarations. The external declarations form one kind of incomplete declarations. The other kind is the *forward* declaration, described in §8.2. External declarations may be used to extend the scope of a declaration in one compilation to another compilation. Forward declarations may be used to extend the scope of a subprogram backwards from its complete definition so as to permit mutual recursion. They may also be used for documentation. It is not necessary that there be a complete declaration corresponding to a given incomplete declaration if the entity created by the definition is never referenced in the text of a compilation.

Each execution of a complete declaration creates an *instance* of the declared entity (a variable, type, or constant). Likewise, every execution of the text forming a defining scope creates an instance of that defining scope, which vanishes when that execution of the defining scope completes. The *extent* of an instance of a declaration is the period of execution time during which that instance exists. This extends from the execution of the declaration itself until exit from the instance of the defining scope during which the declaration was executed. Outer declarations each have a single instance whose extent ends only upon termination of the entire program.

Constant declarations define identifiers and operators to the left of the equals (`=`) signs to denote the values (subprograms are also values) to the right of the equals signs. Values are computed when the declaration is executed. When the defining expression is a composite object (a map or record), the left side may be a list of identifiers in parentheses, which are ascribed the values of the components of the defining expression according to position. There must be exactly as many identifiers as components.

The declarations in a constant declaration list may either be *opaque* or *transparent*, depending on the defining expression. A declaration whose defining expression is a subprogram literal is transparent; all others are opaque.

Informally, opaque declarations hide previous declarations of the same identifiers, while transparent declarations introduce possible interpretations of identifiers without hiding previous ones. Variable declarations and formal parameter declarations are also opaque. §6.1 defines these terms more precisely.

A type declaration defines one or more identifiers to denote types. Type declarations are opaque. §7 describes the possible definitions for such identifiers.

Examples

```
let
    nd = 3,            /* Simple scalar */
    Force = [1 .. nd] long real,    /* Type */
    (x0, x1, x2) = V;     /* Where V is a vector dimensioned 1..3 */
```

6.1 Resolving ambiguity

When an instance of an identifier appears in the scope of more than one declaration of that identifier, there are two ways of resolving the resulting ambiguity. First, certain declarations *hide* previous ones due to opacity. Second, the type-consistency rules (which govern, for example, the required types for subprogram parameters) restrict the set of admissible interpretations of an instance.

An opaque declaration of an identifier (as defined in §6) hides any preceding declaration of that identifier throughout the scope of the opaque declaration. As an aid to catching certain kinds of error, an opaque declaration may not hide a declaration in the same defining scope.

After considering opacity, there may still be multiple declarations covering a particular instance—in which case the declared identifier or operator is said to be *overloaded.* A choice of declarations for all the instances in an expression is called an *interpretation.* An interpretation is *legal* if it obeys the type rules of the language (e.g., the rule that an actual parameter of a function must have the type indicated by the corresponding formal parameter). There must be at least one legal interpretation, and the set of legal interpretations must be such that the types of the expression and all of its subexpressions are uniquely determined. The interpretation chosen is the one that applies the first applicable declaration in order of appearance to each instance.

6.2 Subprogram and Operator Declarations

Syntax

<operator> ::=
    <predefined operator>
  | <other operator>
<other operator> ::=
    <identifier>
  | <operator character> [<operator character> [<operator character>]]
  | '(' <operator character> [<operator character>] ')'
<operator character> ::=
    '*' | '/' | '+' | '-' | '<' | '>' | '='
  | '@' | '#' | '%' | '|' | '&' | '~' | '^'
<operator declaration> ::=
    prec <predefined operator> **operator** { <other operator> ',' ...}$^+$;
  | **postfix operator** { <other operator> **,** ...}$^+$;

Semantics

An ordinary subprogram is declared by a constant declaration whose right-hand side is a subprogram literal. A subprogram's designator may be an operator, in which case calls on the subprogram take the form of expressions in which the designator acts as a prefix, infix (binary), or postfix operator. Prefix and postfix operators must be declared with one argument; infix (binary) operators must be declared with two. Declarations of operators as infix versus prefix operators are distinguished by the numbers of arguments. When an identifier is an operator, it is reserved for use as an operator only. In a given compilation, an operator may be overloaded to be both an infix and a prefix operator but no other combination is allowed.

Any 'other operator' used in a program must be declared in an operator declaration before its first appearance in either an expression or a subprogram designator spec. All such operators are left associative and have the same precedence as the predefined operator that appears in their operator declaration. An operator may appear in any number of operator declarations within a single compilation, but must be given the same precedence in each. All postfix operators have the highest precedence.

Chapter 7

Data Types and Type Constructors

Syntax

<type> ::=
    <user-defined type identifier>
  | <basic scalar type>
  | <record type>
  | <domain type>
  | <map type>
  | <subprogram type>

Semantics

FIDIL supports several classes of data type, and allows the programmer to define new types within some of those classes. The following sections describe the type denotations. Identifiers defined to be types may be used where type names are allowed (e.g., to define variables or the types of parameters.)

In several places, we will refer to two types as being *equivalent*. A type is always equivalent to itself. The following sections define type equivalence for particular classes of types. We will also refer to types being *assignment compatible*. We say that a type T_1 is assignment compatible with type T_2 if values of type T_1 are allowed to be assigned to objects of type T_2, or passed as value parameters to formals of type T_2. Equivalent types are always assignment compatible. The following sections define assignment compatibility for each type class.

7.1 Basic Scalar Types

Syntax

<basic scalar type> ::=
  [**long**] **integer**
| [**long**] **real**
| [**long**] **complex**
| **logical**
| **char**

Semantics

The basic scalar types are the familiar numeric, logical (true/false valued), and character types. The meaning of the qualifier "**long**" is implementation-dependent. In the case of real and complex types, it is intended to correspond to double-precision declarations in FORTRAN. It is not necessary for long quantities to have a precision or range different from non-long quantities.

Two scalar types are equivalent if and only if their denotations are equivalent. A short scalar type is assignment compatible with the corresponding long type (but not the reverse).

7.2 Record Types

Syntax

<record type> ::=
  struct '[' { <field group> ';' ...}+ [';'] ']'
<field group> ::=
  <type> { <identifier> ',' ...}

Semantics

A value or object having a record type is composed of fields as indicated in the definition of the record type. If $T_1, \ldots, T_n$ are type denotations, then

```
let R = struct [ T_1 x_11, x_12, ...; ... T_n x_n1, ... ]
R x;
```

defines R to be a record type and x to be a single variable of that type. Each R consists of fields, which are accessed by (prefix) selectors named x_{ij}. The syntax is identical to that for function calls. Like subprograms in general, record field selector names may be overloaded.

A record type M is equivalent to type M' if and only if M' is a record type with the same number of fields in the same order with the same names, and the type of each field of M is equivalent to the corresponding field's type in M'. Only equivalent record types are assignment compatible.

7.3 Map and Domain Types

Syntax

<domain type> ::=
    [<domain qualifier>] **domain**
  | [<domain qualifier>] **domain** '[' <*integer* expression> ']'
<map type> ::=
    [**flex**] <unspecific map domain> <type>
  | <specific map domain> <type>
  | **valtype** '(' <*domain* expression> ')'
<unspecific map domain> ::=
    [<domain qualifier>] '[' ['*' <*integer* expression>] ']'
<specific map domain> ::=
    [<domain qualifier>] '[' <*domain* expression> ']'
  | <rectangular domain constructor>
<domain qualifier> ::=
    rect

An n-dimensional *domain* is a subset of $\mathbf{Z}^n$, and has a *domain type* denoted **domain**[n] We call n the *arity* of the domain. An *element* of a one-dimensional domain is an integer; an element of an n-dimensional domain, for $n > 1$, is itself a map with type `[1..`n`]` **integer**; that is, it is a tuple of n integers. The notation **valtype**(D) is an abbreviation of the type

`[1..`n`]` **integer**, where $n > 1$ is the arity of D, or of the type **integer** if $n = 1$. Two domain types are equivalent if their arity is identical.

A *map* is a mapping from a domain to some codomain; map types are an extension of ordinary array types. Two map types are equivalent if their domains have the same arity, they have the same flexibility (indicated by the presence or absence of the keyword **flex**), and their codomain types are identical. Two map types are assignment compatible if both domains have the same arity and the codomain types are equivalent. A map type is said to be *unspecific* if it is not flexible and either has an unspecific map domain or an unspecific codomain. An unspecific domain is denoted `[]` (one-dimensional) or `[*`n`]` (n-dimensional). Record types are unspecific if they have at least one unspecific field type.

The modifier **flex** indicates that variables of the type have a modifiable domain. If M is a map type denoted by

flex `[*`n`] T`

and X is a variable of type M, then it is legal to assign to domainOf(X) and it is legal to assign to X a map with a domain differing from that of X. In general, T may be a type with map components that themselves have unspecified domains. An assignment to X will set those domains as well as the domain of X. The domain of an individual component of X, however, cannot be changed independently of the rest of X, unless that component has a flexible type itself (see the examples). A variable declared to have a flexible type initially has all flexible domains (of it and its components) initialized to the empty domain. A variable may not be declared with an unspecific type. Unspecific types are mainly intended, instead, for use in formal parameter specifications.

A *domain qualifier* is an assertion about the values of domains—either the values of domain variables or expressions, or the domains of map variables or map expressions. It does not affect type compatibility. It is an error for the value of a variable, parameter, or expression to violate the assertion denoted by a domain qualifier.

The presence of the **rect** qualifier indicates domains (or domains of map values) that are rectangular subsets of integer tuples.

Examples

```
let
    FlexTabType = flex []  struct [ [] integer A; integer B ],
    FlexRectType = flex [] [] integer,
    RaggedType = [] flex [] integer,
    FlexRaggedType = flex [] flex [] integer,
    PairRaggedType = [1..2] flex [] integer,
    PartRectType =  [1..3] [] integer,
    ARectType = [1..10] [0..9] integer,
    F = proc (ARectType X; ref RaggedType Y)  : ···;

FlexRectType Q;
FlexRaggedType R;
ARectType S;

/* RaggedType T;    ILLEGAL (unspecific domain) */
/* PartRectType U;  ILLEGAL (unspecific domain in codomain) */
PairRaggedType V;

Q := [ [1,2,3], [7,8,9] ];
Q := [ [0,1], [9,10] ];
/* Q := [ [0,1], [7,8,9] ];  ILLEGAL ASSIGNMENT (not rectangular) */
R := [ [0,1], [7,8,9] ];
```

7.4 Subprogram Types

Syntax

<subprogram type> ::= <explicit *subprogram* header>

Semantics

Explicit headers (see §8.2) supply the names and types of formal parameters to a subprogram literal. When used as types, they match any subprogram of the same class taking the same number and equivalent types of arguments, and returning an equivalent type.

Chapter 8

Expressions and Statements

Syntax

<statement> ::=
<expression>
| <control statement>
| <assignment>
<expression> ::=
<subprogram literal>
| <expression2>
expression2 ::= <primary>
| <*prefix* operator> <expression2>
| <expression2> <*infix* operator> <expression2>
<primary> ::=
<number>
| <string literal>
| **true**
| **false**
| <identifier>
| **proc** <operator>
| <subprogram closure>
| <constructor>
| <primary> <*postfix* operator>
| <indexed expression>
| <control expression>

```
        |  ( <expression> )
<predefined operator> ::=
           ** | @ | <@>
        |  * | / | rem | mod | div | <<
        |  + | - | (+) | #
        |  = | < | > | <= | >= | /= | in
        |  and | or | not | on
```

As given, this grammar is ambiguous; the ambiguity is resolved by priority and grouping rules. In the listing above, the priorities of operators listed on the same line are identical; those of operators listed on later lines decreases. Operators group to the left, except for '**', which groups to the right.

Semantics

The primaries **true** and **false** denote the logical constant values.

The primary "**proc** <operator>" denotes the subprogram denoted by the specified operator. (It is used in contexts where, for example, the binary function '+' is to be passed as a parameter.)

The following sections describe the other kinds of expressions and primaries.

8.1 Control Expressions and Statements

Syntax

```
<control expression> ::=
           <if expression>
        |  begin <block> end
<control statement> ::=

        |  <loop statement>
        |  <exit statement>
<if expression> ::=
           if <guard> then <block>
            { elsif <guard> then <block> ...}
             [ else <block> ]
```

fi
<guard> ::=
<*logical* expression>
| <*logical map* expression>
<loop statement> ::=
[<loop type> <control clause>]
do <block> **od**
<loop type> ::=
for
| **forall**
<control clause> ::=
[<control identifiers clause> **from**] <*domain* expression>
[**by** <*integer vector* expression>]
<control identifiers clause> ::=
<identifier>
| '[' { <identifier> ',' ...}$^+$ ']'
<block> ::=
{ <block clause> ';' ...}$^+$ [';']
<block clause> ::=
<constant declaration list>
| <variable declaration>
| <statement>
<exit statement> ::=
return [<expression>]
| **exit**

Semantics

The terms *control expression* and *control statement* generally refer to constructs whose components are not necessarily evaluated in applicative order.

An *if-expression* provides for conditional evaluation; it selects values from its constituent blocks depending on the values of its guards. The guards may either all be **logical** expressions, or they may all be maps with identical domains and codomain **logical**.

When its guards are **logical** expressions, evaluation of an if-expression consists of evaluating the guards in order until they are exhausted or one evaluates to **true**. The corresponding block is evaluated and its value becomes the value of the if-expression. If none of the logical expressions evaluates to **true**, the block following **else**, if any, is evaluated and supplies the value of the if-expression. Otherwise, the if-expression has a void value. If the expression occurs in a context requiring a non-void value, then all the blocks must yield values of the same type, and it is an error for the if-expression to yield a void value.

When its guards are all maps with codomain **logical**, an if-expression's blocks must also be maps whose domains have the same arity as those of its guards, and whose codomains must be identical. The result of evaluating

$$\textbf{if } C_1 \textbf{ then } E_1 \ \ldots \ \textbf{elsif } C_n \textbf{ then } E_n \textbf{ else } E_d \textbf{ fi}$$

is a map M whose domain is the union of the domains of the C_i and whose codomain is the same as the E_i. If p is in the domain of M then $M[p]$ is computed by

$$\textbf{if } C_1[p] \textbf{ then } E_1[p] \ \ldots \ \textbf{elsif } C_n[p] \textbf{ then } E_n[p] \textbf{ else } E_d[p] \textbf{ fi}$$

An **else** clause is required for these if-expressions.

Evaluation of a *loop statement* causes repeated evaluation of the block (which is called the *loop body*.) The loop always yields a void value. In the absence of a control clause, a loop iterates until an exit statement is evaluated.

When a control clause is supplied, the loop body is repeated for each value it specifies, which are bound in turn to the control identifiers, if present. The loop expression declares any control identifiers, whose scope is the loop itself. The control clause specifies a (possibly null) domain (a subset of $\mathbf{Z}^n$ for some $n > 0$.) If there is a single, unbracketed identifier, it is bound to the elements of this domain in some sequence. When the control identifier clause is a bracketed list of identifiers, $[i_1, \ldots i_n]$, then n must be the arity of the domain and for each element, p, of the domain, in some order, the i_j are bound to $p[j]$ (so that $p = [i_1, \ldots, i_n]$).

The control clause determines the domain as follows. The domain expression must have a value, D, of type **domain**`[`n`]` for some n (compatible with the control identifiers clause.) The array expression following the keyword **by**, if present, must have a value, S, of type `[1..`n`]` **integer** or, if $n = 1$, of type **integer** (which we'll treat as a one-element array in what follows). The elements of S must all be positive. If this array expression is not present, S defaults to an array whose elements are all 1. Informally, the array expression specifies a step size for each of the indices of the domain. More rigorously, the domain specified by the control clause is given by the following expression.

$$D * \mathrm{shift}(\mathrm{inject}(\mathrm{project}(\mathrm{shift}(D), S), S), \mathrm{lwb}(D))$$

If the loop type is **for**, then the iterations of the loop body happen sequentially. If the domain expression is a rectangular domain constructor, its elements are enumerated in lexicographic order (last index varying most rapidly in increasing order). Otherwise, no order is guaranteed. When the loop type is **forall**, the iterations of the loop proceed collaterally (possibly in parallel.) Nothing can be guaranteed about the effects of multiple iterations writing to the same variable, or in general about the order of any side-effects.

Evaluation of a block causes the evaluation of its constituent statements in order. This evaluation may be interrupted by the evaluation of an exit statement. The value of a block is that of its last clause (or void if the last clause is a declaration.) A block is a defining scope.

Exit statements provide means of leaving a loop or subprogram body. Evaluating an **exit** terminates the innermost enclosing loop expression. Evaluating a **return** terminates the evaluation of the innermost dynamically enclosing subprogram body, yielding the value of the given expression, which must be of the type returned by the subprogram, as the value of that body. No expression may be supplied for a procedure (a subprogram with a void value.) An exit statement may not cause exit from a **forall** loop.

Examples

```
for [1 .. N] do S od;  /* Repeat S N times. */

for i from [1..N] do S := A[i] + S od;
        /* Add elements 1 to N of A to S. */

for [i,j] from [1..N, 1..N] do M := max(M,A[i,j]) od;
for p from domainOf(A) do M := max(M,A[p]) od;
        /* Find the maximum of all elements in A (two ways). */
```

```
for [i,j] from [0..N, 0..N] by [1,2] do S od;
        /* Repeat S for [i,j] at [0,0], [1,2], ..., [1,0], [1,2], ... */
```

8.2 Subprogram Literals

Syntax

<subprogram literal> ::=
  [<pragma>] <subprogram header> : <subprogram body>
<subprogram header> ::= **proc** [({ <formal> ';' ...}$^+$)] -> <type>
  | **proc** ({ <formal> ';' ...}$^+$)
<generic subprogram literal> ::=
  generic ({ <identifier> ',' ...}$^+$) <subprogram literal>
<formal> ::=
  [**ref**] <*formal* type> { <identifier> ',' ...}$^+$
<subprogram body> ::=
  <expression>
  | **external** [<interface clause>] <identifier>
  | **forward**
<interface clause> ::=
  '(' <identifier> ')'

Semantics

A subprogram literal denotes a subprogram, which, when called, performs a computation and may return a value. A subprogram literal is itself a defining scope.

A subprogram generally takes parameters, whose types must be declared in the subprogram header. Parameters may be passed either by *value*—the called routine being passed a copy of the argument—or (as indicated by the keyword **ref**) by *reference*—the formal parameter of the called routine is identified with the actual parameter during execution of the call.

The body of a subprogram indicates the computation to be performed or the object to be returned. When the body is an expression, its type must be assignment compatible with that of any return type specified for the subprogram. For a selector, that part of the expression denoting its value must have the form of an object denotation, and the type of the denoted object must be equivalent to that specified as the return type.

When the body of a subprogram is the clause **external** P, and the subprogram is used in a program, the subprogram identifier P must must be declared with an expression as body elsewhere in the compilation or must be exported with an expression as body from some compilation in that program. The type of P must be that indicated by the header. For external subprograms written in languages other than FIDIL ("foreign" subprograms), the interface clause indicates the language (and thus any necessary calling conventions or parameter conversions).

Subprogram literals whose body is the keyword **forward** may only appear as the expression in a constant declaration. If the constant declaration in which it appears is used in the compilation, then there must be a corresponding complete constant declaration in the same defining scope, with the same identifier, and in which the expression is again a subprogram literal with the same type. There may be more than one incomplete declaration for a given complete declaration, and each incomplete declaration may appear either before or after the corresponding complete declaration.

A *generic* subprogram literal does not directly denote a subprogram, but is rather a template for subprogram literals. The identifiers, called *generic formals*, in the list following the keyword **generic** may stand for types or for the integer constants that appear in unspecific map type specifiers. In effect, a generic subprogram literal stands for all possible subprograms obtainable by substituting types or integers for these identifiers. The body of a generic subprogram literal is not translated—and therefore not checked for legality—at the time it is initially encountered. Indeed, it can't be, since the types of its arguments, and therefore the meanings of subprogram calls within its body, are unknown. Only when an actual call is resolved to the generic subprogram does the substitution of actual types for the generic formals take place, and only then is the resulting body checked for semantic consistency.

The scope of the generic formals is the entire subprogram literal that follows them. They are not hidden by any inner declarations; it is illegal to declare quantities with the same names as the generic formals within the subprogram literal.

8.3 Subprogram Calls and Partial Closures

Syntax

<subprogram closure> ::=
        <*procedure* primary> '(' { <closure argument> ',' ...}$^{+}$ ')'
<closure argument> ::=
        <expression>
     | '?'

Semantics

A subprogram closure first causes evaluation of any arguments. If all arguments to the function are present, then execution continues with execution of the body of the subprogram designated by the procedure primary. In this case, the closure is called a *call*. If $r > 0$ arguments are missing (that is, are left null, with the surrounding commas left in,) then the closure is called a *partial closure* of the designated subprogram. The value of this partial closure is itself a subprogram taking r arguments. At least one argument must be supplied in a partial closure. When this partial closure's value is itself called with r arguments, the result is as if those argument values were inserted for the arguments missing from the closure and the result treated as an ordinary call.

A subprogram with no arguments must have a void value (it is executed for its side-effects alone.) To be called, this subprogram's designator must be followed by empty parentheses.

The order in which argument expressions are evaluated is undefined.

8.4 Constructors

Syntax

<constructor> ::=
        <map constructor>
     | <record constructor>
     | <rectangular domain constructor>
<map constructor> ::=

‘[’ [‘*’ <integer literal> ‘:’] { <expression> ‘,’ ...}$^+$ ‘]’
| ‘[’ <control clause> : <expression> ‘]’

<record constructor> ::=
<*record type* identifier> ‘(’ { <expression> ‘,’ ...}$^+$ ‘)’

<rectangular domain constructor> ::=
‘[’ { <dimension> ‘,’ ...}$^+$ ‘]’

<dimension> ::=
<*integer* expression> .. <*integer* expression>

Semantics

Constructors are expressions that build composite objects. The components specified by the expressions must conform in type and number to the components indicated by this type. Both maps and records may be specified extensionally—as a list of expressions. The n^{th} component expression corresponds to the n^{th} field of the record or to a first index of n for a map. The optional ‘*n:’ at the beginning of a map constructor is used to indicate the arity of the map; it defaults to one. In an extensional map constructor for a map of arity $n > 1$, each component expressions must itself represent a map of arity $n-1$ (except that the ‘*n:’ is unnecessary, and indeed ignored, for these component expressions). The most deeply-nested expressions correspond to the first index. For example,

```
let
    C = [*2: [1, 2], [3, 4], [5, 6] ],
        /* C is of type [*2] integer. */
        /* C[[1,1]] = 1, C[[2,1]] = 2, C[[1,2]] = 3, etc. */
    Q = [ [1, 2], [3, 4], [5, 6] ];
        /* Q is of type [*1] [*1] integer. */
        /* Q[1][1] = 1, Q[1][2] = 2, etc. */
```

A map constructor may also specify a rule (the second form given in the syntax.) The control clause (see §8.1) then specifies the domain of the map. For each item of the domain, the expression is evaluated to give the image of that value. If the control clause specifies control identifiers, these may be used in the expression. Evaluation is as for **forall**: no order is specified and the elements may, in fact, be evaluated in parallel.

A record constructor for a record type, R, whose fields have types T_i may also be applied to arguments whose types are $[D]\ T_i$ for some (common) domain D. The result is of type $[D]R$; its fields' values at a given domain point are those of the arguments at that point. Likewise, such a record constructor may also be applied to arguments of type $[D_1]$ $[D_2]$ T with result of type $[D_1]$ $[D_2]$ R, and so forth.

The domain constructor yields a domain consisting of all tuples of integers such that each integer is in the range denoted by the corresponding element of the dimension list.

Examples

```
[ [1..10] : 0 ]
    /* The constant 0 map on the domain 1..10. */
[ i from [1..10] : i ]
    /* The identity map on the domain 1..10 */
```

8.5 Indexing

Syntax

<indexed expression> ::=
    <*map* primary> { <indexer> ...}$^+$
<indexer> ::=
    '`[`' { [<expression>] '`,`' ...} '`]`'

Semantics

The notation "$A[\alpha_1, \ldots, \alpha_n]$" is equivalent to "$A[\alpha_1] \cdots [\alpha_n]$." In the following discussion, we assume the latter form.

Indexing a map produces the value or (assignable) object in the codomain of a map corresponding to a given element of the domain. The expressions may be integers, in which case the domain value is the tuple consisting of those integers, or there may be a single expression in the domain of the map. It is an error if the index is not in the current domain of the map.

An expression E of the form

$$A\underbrace{[\,]\cdots[\,]}_{n}[i_n],$$

is a selector denoting a map object with the property that

$$E[i_0]\cdots[i_{n-1}] \equiv A[i_0]\cdots[i_n]$$

(i.e., these two expressions denote the same variable; taking their value or assigning to them have identical results.)

8.6 Assignment

Syntax

<assignment> ::=
        <left side> `:=` <right side>
    | <left side> `*:=` <right side>
<left side> ::=
        <*object* expression>
    | ({ <*object* expression> ',' ...}$^+$)
<right side> ::=
        <expression>
    | ({ <expression> ',' ...})

Semantics

An <*object* expression> in the syntax above refers to an expression that designates an (assignable) object (a variable, indexed expression, or selector call). An assignment may be to one or to several such objects simultaneously. In the latter case, the right side may be either a single, record-valued or map-valued expression—whose components are assigned in order to the objects on the left side—or a parenthesized list of values. There must be identical numbers of values to be assigned as objects to receive them.

The '`:=`' operator assigns an entire object of any type. The types of the right-side entities must be assignment compatible with those of the corresponding left-side objects. A flexible map object may receive a map value with any domain of the appropriate arity. A non-flexible map object may only receive maps with identical domains. These rules apply recursively to the elements of the codomain.

The '`*:=`' operator applies only to maps. The rules governing it are the same as for '`:=`', except that it assigns values from the right side(s) only at points in the intersection of the domains of the left and right sides.

8.7 Standard Functions and Operators

Tables 8.1 and 8.2 give the standard operators and functions on domains. Tables 8.3–8.6 give the standard operators and functions on maps. Tables 8.7 and 8.8 give the standard arithmetic and mathematical operators.

Expression	Meaning
nullDomain(n)	The empty domain of type **domain**[n]. The quantity n must be a compile-time integer constant.
$D_1 + D_2$	Union of D_1 and D_2.
$D_1 * D_2$	Intersection of D_1 and D_2.
$D_1 - D_2$	Set difference of D_1 and D_2.
p **in** D	where D is a domain of arity n and p is an array of type **valtype**(D): a logical expression that is **true** iff p is a member of D.
lwb(D) upb(D)	For a domain of arity n: An **integer** map with domain $[1..n]$ (for $n = 1$, an **integer**) whose k^{th} component is the minimum (lwb) or maximum (upb) value of of the k^{th} component of the elements of D.
arity(D)	yields n for a domain of arity n.
sizeOf(D)	The cardinality of D.
shift(D, S), $D << S$	Where S is of type **valtype**(D) and n is the arity of D: The domain $\{d + S \| d$ **in** $D\}$.
shift(D)	Same as shift(D, -lwb(D)).
inject(D, S)	The domain $\{d{*}S \| d$ **in** $D\}$.
project(D, S)	The domain $\{d \oslash S \| d$ **in** $D\}$, where '$\oslash$' denotes elementwise integer division, rounding toward $-\infty$.
expand(D, S)	The domain $\{e \| e \oslash S$ **in** $D\}$.
contract(E, S)	The domain D such that $E =$ expand(D, S), if it exists.

Table 8.1: Operators and functions on domains, part 1.

Expression	Meaning
accrete(D)	The set of points that are within a distance 1 in all coordinates from some point of D.
boundary(D)	accrete(D) $- D$.
reduce(D, S)	where D is a domain of arity n and $S = [i_1, \ldots, i_r], 1 \leq i_1 < \ldots < i_r \leq n$. The result is the domain of arity $n-r$ consisting of all $(n-r)$–tuples $[j_1, \ldots, j_{i_1-1}, j_{i_1+1}, \ldots]$ such that $[j_1, \ldots, j_n] \in D$.

Table 8.2: Operators and functions on domains, part 2.

Expression	Meaning
domainOf(X)	The domain of map X. This may also appear in a left-hand side context if X is a partial map variable. The result of an assignment to the domain of X is a map whose initial image consists of undefined values.
toDomain(X)	where X is a **logical** map: $\{p \in \text{domainOf}(X) \| X[p]\}$.
image(X)	where X is a map whose codomain is an integer map of arity n: the domain of dimension n whose elements are all elements in the image of X—that is, the set $\{d \| X[p] = d, \text{ for some } p\}$.
upb(X) lwb(X) arity(X)	upb(domainOf(X)) lwb(domainOf(X)) arity(domainOf(X))
X `#` Y	The composition of X and Y. X and Y are maps; Y's codomain must be **valtype**`(domainOf(`X`))`; and `image(`Y`)` must be a subset of `domainOf(`X`)`. X `#` Y is a map object (which is assignable if X is assignable) such that $(X\#Y)[p] \equiv X[Y[p]]$. Hence, its domain is `domainOf(`Y`)`.
shift(X, S), $X << S$ shift(X)	where S is a $[1..n]$ **integer** (an integer for $n = 1$), with default value -lwb(X), and n is the arity of X: the map `X # [p` **from** `domainOf(`X`): p-`S`]`.
inject(X, S)	`X # [p` **from** `inject(domainOf(`X`), `S`): p/`S`]`.
project(X, S)	`X # [p` **from** `project(domainOf(`X`), `S`): `S`*p]`.
contract(X, S)	`[p` **in** `expand([0..0, ..., 0..0], `S`) :` `[project(`X `<< -p, `S`)]`.
expand(X, S)	Produces a map defined by the relation `expand(contract(`X, S`),`S`) = `X.

Table 8.3: Operators and functions on maps, part 1.

Expression	Meaning
X **on** D	The map X restricted to domain D.
X (+) Y	where domainOf(X) $\cap$ domainOf(Y) $=$ $\{\}$: the union of the graphs of X and Y, whose codomains must be identical and whose domains must be of identical arity.
concat(E_1,...,E_n)	Concatenation of $E_1, \ldots, E_2$. The E_i must be 1-dimensional maps with contiguous domains and some (common) codomain T, or values of type T, which are treated as one-element maps with lower bound 0. At least one of the E_i must be a map on T. The result has the same lower bound as E_1 and an upper bound equal to the sum of the lengths of the E_i.
F`@`	Assuming that F takes arguments of type T_i and returns a result of type T, F`@` is a function extending F to arguments of type $[D_i]$ T_i, where the D_i are domains of the same arity, and returns a result of type $[D]$ T, where D is the intersection of the D_i. The result of applying this function is the result of applying F pointwise to the elements corresponding to the intersection of the argument domains.
F `<@>`	For F as above returning type T_1: The extension of F to arguments of types $[D_i]T$ as above, returning a value of type $[D_1]T_1$ defined by $F\!<@>(x_1, \ldots, x_n) = F@(x_1, \ldots, x_n)\ (+)\ (x_1\ \mathbf{on}\ (D_1 - D))$.

Table 8.4: Operators and functions on maps, part 2.

Expression	Meaning
compress(X)	where X is a map on a domain of arity 1: The one-dimensional map, X' with a contiguous domain having a lower bound of 1 such that $X'[i]$ is the value of $X[p_i]$, for p_i the i^{th} smallest element in the domain of X.
compress(X, W)	where W is a one-dimensional map whose codomain is **logical**: compress(X **on** toDomain(W)).
decompress(X, W)	The map X' such that compress$(X', W) =$ compress(X).
reduce(X, f, S, v_0)	where X is a map of arity n and codomain C; $S = [i_1, \ldots, i_r], 1 \leq i_1 < \ldots < i_r \leq n$; and f is a function taking two arguments, one of some type R, the second of type C, yielding a result of type R. The result, B, is of type $T = [*(n-r)]R$, or $T = R$ if $n = r$, and has domain reduce(domainOf(X),S). Its values are defined as follows. $B[j_1, \ldots, j_{i_1-1}, j_{i_1+1}, \ldots]$ $= f(f(\cdots f(v_0, v_1), \cdots), v_m)$. where the v_i are the elements $X[j_1, \ldots, j_{i_1-1}, k, j_{i_1+1}, \ldots]$ for all k for which the expression is defined, taken in some undefined order.
reduce(X, f, v_0)	where X is any map with codomain C; v_0 is of some type R; and f is as above. The result is of type R and has the value v_0 if the domain of X is empty, and otherwise $f(f(\cdots f(v_0, v_1), \cdots), v_m)$ where the $v_i, i > 0$ are the elements of X is some undefined order.

Table 8.5: Operators and functions on maps, part 3.

Expression	Meaning
sort(X, P)	where X is a contiguous, one-dimensional map with codomain T and P is logical-valued binary function with arguments of type T: the map X' with the same domain as X that results from permuting the image of X so that $i < j$ implies $P(X'[i], X'[j])$. The permutation is strict: the order of image elements x and y such that $P(x, y)$ and $P(y, x)$ is unchanged by the sort.
trace(A, S)	reduce(A,**proc +**, S, 0)
outerproduct(A, B)	where A and B are maps with rectangular domains of dimensions n_a and n_b and the same codomains: The map C defined as follows. $C[i_1, \ldots, i_{n_a}, j_1, \ldots, j_{n_b}] = A[i_1, \ldots, i_{n_a}] * B[j_1, \ldots, j_{n_b}]$
transpose$(X[, \pi])$	where $\pi = [\pi_1, \ldots, \pi_n]$ is a permutation of the integers between 1 and n, and n is the arity of the map X: The object, X', resulting from transposing the indices of X according to π. Specifically, $X'[i_{\pi_1}, \ldots, i_{\pi_n}] = X[i_1, \ldots, i_n]$. The default for π is [2,1].
flip(X, π)	where X is of type $[D_1] \ldots [D_n]\ T$: The map, X' defined by the following. $X'[p_{\pi_1}] \ldots [p_{\pi_n}] = X[p_1] \ldots [p_n]$. The default for π is [2,1].
flip(X)	where X is a record of maps with identical domains: produces the map taking p in the common domain to the record with field values $F_i[p]$, where the F_i are the fields of X. X can also be a map of records, in which case flip performs the inverse operation.
remap(X)	The object resulting from "reassociating" the indices of X, which must be of type `[*`m`][*`n`]`T to form an isomorphic object, Y of type `[*`$m+n$`]`T. If p is a valid index of X and q is a valid index of $X[p]$, then Y`[concat(`p, q`)] = `X`[`p`][`q`]`.
remap(Y, m)	If X, Y, and m are as above, then `remap(`Y, m`)=`X.

Table 8.6: Operators and functions on maps, part 4.

Expression	Meaning
`+`, `-`, `*`, `/`, `**` **rem**, **mod**, **div**	For scalar arguments, the standard arithmetic operators. The division operator, '`/`', produces a **long real** result when its operands are integers. The binary operator **div** applies to integers, producing the quotient of its operands truncated to an integer. The **rem** operator is defined by the formula $x = (x \textbf{ div } y) * y + (x \textbf{ rem } y)$, for $y \neq 0$. The **mod** operator is defined by $x \textbf{ mod } y = x - y \lfloor x/y \rfloor$, for $y \neq 0$. The same conversion rules apply as for FORTRAN. When applied to maps of the same arity and codomain, these operators apply pointwise, producing a map whose domain is the intersection of the domains of operands. Finally, the operators are also overloaded to allow one operand to be of a scalar type T and the other to be a map whose codomain has a type that the operator can legally combine with type T. In this case, the operand of type T is treated as a constant map with the same domain as the other operand. This latter definition is recursive; for example, the codomain of the map operand may itself be a map.
<, >, <`=`, >`=`, `=`, `/=`	Relational operators (`/=` is "not equal.") These operators also extend to maps as for the arithmetic operators.
and, **or**, **not**	The standard logical connectives. These also extend to maps of scalars.

Table 8.7: Arithmetic Operators and Elementary Functions, part 1.

Expression	Meaning
$\exp(x)$, $\ln(x)$, $\log10(x)$ $\sin(x)$, $\cos(x)$, $\tan(x)$ $\mathrm{sqrt}(x)$ $\mathrm{atan}(x)$, $\mathrm{atan}(x, y)$	The standard elementary mathematical functions. They are defined on real and complex quantities, yielding results of the same type.
$\mathrm{abs}(x)$	Absolute value. For real and complex quantities, yields a real value of the same length, otherwise an integer.
$\mathrm{floor}(x)$, $\mathrm{trunc}(x)$, $\mathrm{round}(x)$, $\mathrm{toSingle}(x)$, $\mathrm{toLong}(x)$, $\mathrm{toInt}(x)$ $\max(x_1, \ldots, x_n)$ $\min(x_1, \ldots, x_n)$	Scalar coercions. Floor, trunc, and round apply to reals, producing results rounded toward $-\inf$, toward 0, and toward nearest. The functions, toInt, toSingle, and toLong apply to all types, converting to the nearest integer, single-length real (complex), or long real (complex) quantity. The last three operations also act on **logical** values, converting **true** to 1 or 1.0 and **false** to 0 or 0.0. Maximum and minimum. All operands must be of the same type—an integer or real type.
$\mathrm{signum}(X)$	Returns the integer -1, 0, or 1, depending on whether X (which may be an integral or real) is negative, zero, or positive.
$\mathrm{realPart}(Z)$, $\mathrm{imagPart}(Z)$	Real and imaginary parts of the complex quantity Z. Either of type **real** or **long real**, depending on the the type of Z.

Table 8.8: Arithmetic Operators and Elementary Functions, part 2.

Chapter 9

Pragmas

A *pragma* is an "escape clause" allowing the programmer to give the translator advice or other directives that have no semantic effect or that do not fit naturally into the rest of the language.

Syntax

<pragma> ::= '`(*`' { <*pragma* expression> '`,`' ...}$^+$ '`*)`'

Semantics

The possible pragma expressions are given in Table 9.1. The interpretation of a pragma expression depends on the particular pragma; it need not follow the usual strictures of FIDIL semantics.

Expression	Meaning
inline	Indicates that calls on the subprogram literal to which this pragma is attached should be open-coded (that is, each call should be replaced by a suitably-modified copy of the body).

Table 9.1: Pragmas.

Index