Spice language manual

Chris Dollin
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2002-229
October 30th , 2002*

Spice is a programming language suitable for occasional programmers working with text, HTML, and XML, while scaling to medium-sized projects with several experienced programmers. It is an expression-oriented language with multiple values, dynamic typing, automatic storage management, and multiply-dispatched methods. This document contains a general introduction to the language by example, suitable for readers with some programming experience, and a reference manual describing the language in detail.

* Internal Accession Date Only
Approved for External Publication

Contents

Part I

preamble

Chapter 1

origins, growth, and features

This document describes the programming language Spice, which was developed, from an idea of Dave Raggett's, by Chris Dollin and Steve Leach. Spice is intended to be a programming language suitable for occasional programmers working with text, HTML, and XML, while scaling to medium-sized projects with several experienced programmers.

The initial design work was done in autumn 1998 and subsequent modifications made as implementations progressed and the designers gained some experience with their creation.

The original language contained stylesheets and its syntax was closer to ECMAScript. The current language has lost stylesheets, which we could not fit cleanly into our general framework, and its syntax has no C-like basis, being more like that of Pop11. The core of the language language remains and we are experimenting with the addition of regular expressions.

The designers had some particular properties that they wished to exploit together in a programming language.

Spice has *automatic storage management*; programs allocate new values, which are disposed of automatically of when they are no longer referenced. This frees the programmer from a considerable administrative burden.

Spice is *dynamically typed*; by default, types are run-time entities attached to values, rather than compile-time entities attached to expressions. This gives flexibility in program development, at the cost of reduced up-front error detection and some run-time overhead. Type annotations can be added to variable declarations, and the compiler will use these for error checking and optimisation.

Spice is *expression oriented*; (almost) all of the constructs of the language return

values, including loops. Spice is *multiple valued*; expressions denote a "row" of zero or more values. The number of values an expression denotes is not a compile-time property.

Spice is *higher order*; it supports functions as values, full lexical scoping, and has a syntax for function literals (aka lambda-expressions).

Spice is *multiply dispatched*; functions can be defined piecemeal ("overloaded") on differing combinations of argument types, and when the function is called, the piece with the "best fitting" formal arguments is executed. The usual single-receiver dynamic dispatch of object-oriented languages (Smalltalk, C++, Java) is a special case of Spice's multi-methods, which are inspired by CLOS and ObjectClass.

Spice is *class neutral*; new overloadings of any generic function can appear in any package where that function is visible, and there is no special syntax that distinguishes the invocation of an overloaded function from a non-overloaded one. Classes do not provide namespaces.

The remainder of this document is in two parts; an introductory guide, intended to give a practicing programmer a taste of Spice, followed by a semi-formal reference manual describing the language as of September 2002.

Part II

a gentle introduction to Spice

Chapter 2

a gentle introduction to Spice

In this and subsequent sections we speak as though the reader has access to a Spice system into which they can type Spice code and have it executed. Lss fortunate readers will have to take the results on trust.

2.1 hello, world

A long-standing Unix-spawned tradition is that one's first program in a new language should be the one that prints "Hello, World" or some suitable variant, so here it is in Spice:

```
# the traditional example
println( "hello, world." );
```

Pasting this into your Spice evaluation should result in "hello, world" being displayed somewhere obvious.

The line starting "#" is a Spice comment; the "#" and the following characters, up to the end of the line, are ignored by the compiler. A lone "#" anywhere in a line (unless as part of a string) introduces such a comment.

"hello, world." is a *string literal* representing a sequence of characters. Most characters can appear as themselves in a string, but specific exceptions are all three quoting characters (*string quotes*, *character quotes* ', and *symbol quotes* `) and the *escape character* \.

println is the name of a built-in *procedure*. We'll see later that Spice has several kinds of procedure, including *functions*, *methods*, and *constructors*; we use the

term "procedure" to refer to them all without distinction.

The syntax **F(X)** is one form of *procedure call*; it evaluates the procedure **F** (which in this case is easy) and the argument(s) **X** (ditto) and then *calls* the procedure, supplying it with the values of the arguments. The procedure does something (such as printing its argument) and may return some results. **println** doesn't have any results.

The semicolon is a *statement separator*, signifying the end of one statement and (possibly) the beginning of another. Sometimes they are not necessary, but this introduction will put them all in.

2.2 one, two, buckle my shoe

println isn't restricted to strings; it can do numbers, too.

```
println( 1 );
println( 1_032 );
println( 40 + 2 );
```

Evaluating these should result in the values **1**, **1032**, and **42** being printed. The underbar in **1_032** is just a visual separator. Spice has the "usual" arithmetic operators, +, **-**, *****, and /, although they have some extra wrinkles we'll meet later.What's more, **println** can take multiple arguments:

```
println( 1, 2, "buckle my shoe" );
```

which will print **1 2 buckle my shoe**. Note that the arguments are printed preceeded by a space (this behaviour can be changed), and that there's only one newline printed, at the end. The degenenerate case **println()** with no arguments just prints a newline.

2.3 variables

You can declare[1]variables to hold values.

```
var x := 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;
println( x );
```

which will print **55**. The value **x** holds can be changed by assignment:

[1]variables *must* be declared; trying to use a name that hasn't been declared is an error, even if you're assigning to it, unlike eg Basic or Javascript.

```
x := x * 10; println( x );
```

will print **550**. Using the word **val** instead of **var** makes an *unassignable* variable (rather an oxymoron, but there it is).

2.4 typed variables

It's possible to specify that a variable is restricted to holding values of a given type. For example,

```
var count: Small := 0;
someActionsHere();
count := "of Monte Christo";
```

declares **count** as a variable of type **Small**, restricted to "small" integer values (typically in the range plus-or-minus half-a-billion) and then throws an error when it attempts to assign a string (of type **String**) to it. If a variable is declared with a type but no initial value, then the *default value* for that type is used to initialise the variable; the default value of type **Small** is **0**:

```
var anInt: Small; println( anInt );
```

will print **0**. If there's no type then the variable is declared to be of type **Any** (which means what it sounds like); the default value of **Any** is a special value called **absent**.

Chapter 3

function definitions

There's no magic about procedures that the user cannot harness. A simple way to define your own procedure is to write a **function** definition:

```
define function add( x, y ) => x + y enddefine;
println( add( 1, 2 ) );
```

add is defined and then invoked in the usual way, resulting in **3** being printed; the body of **add** is evaluated[1]to deliver its result.

3.1 typed arguments

The arguments to a function can be typed in the same way that variables can.

```
define function addChecked( x: Small, y: Small ) =>
    x + y
enddefine;
```

Calls to **addChecked** with non-**Small** arguments will throw an error. Spice can (sometimes) take advantage of the type information to produce better code or more informative error messages. Later on, we'll see how type information can be used to write *polymorphic* or *overloaded* functions.

[1]There's no need for a special **return** statement to specify the procedure's return value; the value of the procedure body is returned as the procedure result, and the body here is the sum-expression **x+y**.

3.2 typed results

The result of a function can also be typed.

```
define function addToInt( x, y ) returns Small =>
    x + y
enddefine;
```

The result from **addToInt** must be an **Small** value; if not an error will be thrown. Again, the Spice compiler may be able to use this information to generate better code or error messages.

3.3 indefinite arguments

How does **println** manage to have arbitrarily many arguments? It uses an *indefinite argument*, marked with ellipsis notation:

```
define function gather( rosebuds... ) => rosebuds enddefine;
println( gather( "while", "ye", "may" ) );
```

rosebuds is an indefinite argument whose value is all of the (remaining) arguments to the function. This prints as **{while ye may}**, which is the way *array values* are printed; the arguments are turned into an array.

It's possible to have one or more *definite* arguments before or after a final indefinite argument:

```
define function foo( x, y... ) as
    println( x ); println( y )
enddefine
```

foo needs at least one argument **x** but takes arbitrarily many more, which are gathered into **y**. It then prints both **x** and **y**; if there are no extra arguments, then **y** is an *empty array*, which prints as **{}**. **println** returns no results, so **foo** returns no results.

3.4 abbreviating function definitions

Function definitions are so common that, in the interests of brevity, it's usually possible to omit the word **function** in the definition:

```
define subtract( x, y ) as x - y enddefine;
```

We will usually leave it in.

Chapter 4

arrays

Spice allows arrays to be constructed using *array expressions*, which are expression sequences enclosed in braces:

```
var lots := {27, "now", "is", "the", "time", 42};
    println( lots );
```

which will print **{27 now is the time 42}**. Arrays can contain values of any type. Individual elements can be extracted by indexing:

```
println( lots[2] ); lots[2] := "anguish"; println( lots );
```

which will print **now** and **{27 anguish is the time 42}**[1]. The function **length** delivers the number of elements in its array argument:

```
println( length( lots ) );
```

will print **6**.

[1] The first element of a Spice array has index **1**, so element **2** is the second element.

Chapter 5

dot and infix notation

It's often useful and clearer to write calls to one-argument functions, expecially those that just extract components from structured values like arrays, using *dot notation*:

```
println( lots.length );
# or even:
lots.length.println;
```

are just other ways of writing the previous calls. Similarly it is often clearer to write calls to two-argument functions using *at notation*:

```
println( 4 @add 5 );
```

will print **9**, after calling the **add** defined earlier. You can cascade this use of @ just as you can a plain operator:

```
println( 1 @add 2 @add 3 @add 4 );
```

will print **10**.

In fact you can call procedures of *any* number of arguments using dot- or at-notation, because of a general rule of the language:

```
X.f(Y) == f(X,Y) == X @f Y, and X.f == f(X) == X @f
```

When at-notation and dot-notation[1]are used together, the "smaller symbol" (the dot) makes a "smaller expression" than the larger one, so that

[1]The *only* difference between `.` and @ is that `.` is much more tightly binding – it's the most binding infix operator – and @ is loosely binding, being only just tighter than the logical connectives **and** and **or**.

```
x.f @g y.h
```

means

```
(x.f) @g (y.h)
```

5.1 dot and infix function definition

Although the plain fucntion definition syntax we've seen already is enough, because of the equivalence of the different notations we've described, Spice allows you to define a function in the style in which you expect it to be called:

```
define function x @add y as x + y enddefine;
# or
define function x.add (y) as x + y enddefine;
```

The arguments can still be typed using **:**, and for a function defined with dot-notation, the special **on** syntax:

```
define function x:Small @add y:Small as
    x + y
enddefine;
#
define function on Small x.add (y:Small) as
    x + y
enddefine;
```

Apart from the type differences, all four of these definitions are equivalent, and **add** can be called in any of the ways we've discussed.

We'll see later how these notations tie into the more common object-oriented approaches, and why the **on** syntax is so-called.

Chapter 6

ifs and buts

So far, all of our expressions have been unconditional – get the arguments, do something, deliver the result; no choices are involved.

Spice has *conditional expressions* for expression choices, and *relational expressions* for tests.

```
if lots.length < 10 then "short" else "long" endif.println;
```

The relational expression **lots.length** < **10** compares the length of **lots** (which is probably still **6**) with **10** and delivers **true** if it's less than and **false** if it's equal or greater. **true** and **false** are the built-in values of type **Bool**. Spice also has the obvious >, <=, and >=, and = for equality, and the less obvious /= for inequality.

The conditional expression tests the boolean value; if it is **true** its value is its **then** arm, if it is **false** its value is its **else** arm, and if it's not boolean[1], an error is thrown.

println is being called here with dot notation, with its argument being the result of the **if**-expression, here the string **short**.

6.1 elselessness and elseif

The **else** part of a conditional is optional. If you miss it out, it's as though you'd written one with no expression following.

[1] Unlike various other languages, **0**, the null string **""**, and the null reference **absent**, are *not* alternatives to **false**, and values such as **1**, **"yes"**, and **println** are *not* alternatives to **true**.

If you want to write a chain of tests, rather than ending up with ever-more-deeply nested **if-endif** structures, Spice lets you introduce additional tests with **elseif**:

```
if x = "hello" then
    println( "well hello there!" )
elseif x = "goodbye" then
    println( "sorry to see you go" )
else
    println( "eh? what does", x, "mean?" )
endif
```

You can have as many **elseif**s as you need.

6.2 ands and ors

The test of an **if** can be composed using the special operators **and** and **or**. **X or Y** is true if **X** is or if **Y** is, but **Y** is only evaluated if **X** is **false**. Similarly **X and Y** is **true** if both **X** and **Y** are **true**, but **Y** is not evaluated if **X** is false.

That's what makes these operators "special": their right operand is only evaluated if its value is needed.

6.3 short conditionals

Spice also permits "short" conditional expressions, for which it uses the same syntax as Algol68 (by coincidence): the expression **(X | T | F)** is equivalent to **if X then T else F endif**.

6.4 unless and do

Spice has an alternate form of **if**, the **unless** (with closing keyword **endunless**) which is the same except the sense of the (first) test is inverted. You can use the word **do** in place of **then**; although you can do this in any part of an **if** or **unless**, it usually turns up in an **unless** with no **else**.

Chapter 7

loops

Spice has a rich **for** loop construct, which allows iteration over collections, termination on conditions, and specification of loop results. A subset of those is available in the **repeat** construct.

7.1 repeat

Suppose we want to iterate over the elements of an array **lots**. Here's one way to do it with **repeat while**:

```
var i := 0;
repeat while i < lots.length do
    ;;; stuff involving lots[i] ...;
    i +:= 1
endrepeat
```

The expression **i +:= 1** is equivalent to **i := i + 1**, but more compact, especially when **i** is some complex expression.

For example, to find the element of an array containing a specific element by complete linear search:

```
define function findIndex( a, x ) as
    var i := 0;
    var here := 0;
    repeat while i < a.length and here = 0 do
        if a[i] = x then here := i endif;
        i +:= 1
    endrepeat;
```

```
    here
enddefine
```

The index of the first element equal to **x** is stored into **here** and then returned as the procedure result. The equality test = compares values using type-specific code.

If you're wondering why Spice uses **repeat while**, and not just **while**, the reason is that it also has **repeat until** (where the condition is inverted), and that you can write code between the **repeat** and the **while**. **while** is also used in **for** loops.

7.2 for

We can perform the same loop with automatic counting and implicit value return, using a **for**-loop with two *control clauses*:

```
define function findIndex( a, x ) as
    for
        i from 1 to a.length
        until a[i] = x then i
        finally 0
    endfor;
enddefine
```

The variable **i** is automatically declared as a variable that exists only while the loop is executing; it takes values from the **from** value (**1**) to the **to** value (**a.length**) inclusive. The loop terminates early when an element **a**[**i**] = **x** is found, and it delivers the result **i**. If the loop terminates normally (ie by exhaustion of a source), it returns the value of the **finally** clause, here **0**.

for loops can also iterate over collections, using the **in** syntax:

```
define function findIndex( a, x ) as
    for
        i from 1; ax in a
        until ax = x then i
        finally 0
    endfor;
enddefine
```

ax is automatically declared, and takes on the value of each element in turn; we still need to keep track of the index, so we count **i** from 1 upwards anyway. Leaving out the limit for **i** means that it will count upwards "forever"; similarly, leaving out the **from** part starts it at **1**. (No, you can't leave out both.)

Chapter 8

multiple values

So far, we've seen Spice expressions and procedures which deliver *single* results. Spice expressions can deliver *multiple* results directly[1], often avoiding the need to construct new objects to hold them.

```
define function plusOrMinus( x, y ) =>
    (x + y, x - y)
enddefine;
```

The comma keeps both its left and right operand values (exactly as it does in a function argument list); **plusOrMinus** delivers *two* values.

```
println( 20 @plusOrMinus 17 );
```

will print **37 3**.

8.1 one and none

Because Spice's multiple values get everywhere, sometimes you need to prune them. There are two useful built-in functions for this: **none** and **one**.

none(X) simply discards the values of all its arguments, as though it were defined by:

```
define function none(ignored...) as enddefine;
```

[1] Another trick for getting several values out of procedure calls is to use reference parameters - but Spice doesn't have those. The nearest equivalent would be to pass already-made objects and update them.

one(X) returns the first of the argument values **X**, if there are any, and otherwise delivers **absent**, as though it were defined by:

```
define function one(args...) as
    (args.length == 0 | absent | args[1])
enddefine;
```

8.2 multiple-value loops

The result of a loop is all the results from the executions of its body, plus any finally-values.

```
println( for i from 1 to 10 do i*i endfor )
```

to print **1 2 9 16 25 36 49 64 81 100**.

8.3 explode

One built-in function produces multiple results as a matter of course; **explode**. **explode** takes one argument and explodes it into its constituents (if any) as a multiple result. Applied to an array, it explodes it into its elements; applied to a string, it explodes it into its characters. Applied to an atomic object (a number, a character, a boolean, **absent**) it delivers *no* values.

Chapter 9

procedures as values

Spice procedures are not restricted to being defined and called; there are several ways they can be manipulated.

9.1 first-class procedures

Spice procedures are *first-class values*; they can be passed as parameters, returned as results, and stored into variables and data-structures such as arrays.

This is particularly useful when you need to do something to all the elements of a collection, but the details of how the collection are kept is secret (eg to allow you to change it later); you write a function that walks over the collection and does "something" to each element. The only collection type we've met so far is arrays, so here's an example supposing that **anArray** is an array variable:

```
define function appCollection( f ) =>
    for x in anArray do x.f endfor
enddefine
```

Now the expression **appCollection(println)** (which of course can also be written **println.appCollection**) will print all the elements of **anArray**. If the elements *do* something sensible when passed to some function **mangle**, then **mangle.appCollection** will mangle all of them.

appCollection returns all the values that **f** returns for the caller to do with as they like. Of course, we might want to construct a new collection from the old one by mangling each element:

```
define function mapCollection( f ) =>
```

```
    {f.appCollection}
enddefine
```

which gathers up the results from **appCollection** and makes a new array out of them.

9.2 lambda-expressions

Passing functions as arguments, and applying them, is all very well, but it can get tedious defining trivial functions to use. For example, if we want to make a new collection by incrementing every element of the old one, we can write:

```
define function inc( x ) as x + 1 enddefine;
inc.mapCollection.println;
```

If **them** is, say, the collection **{0 8 41}** then this will print **{1 9 42}**. But it's tedious to have to define a function, put it somewhere in the code, give it a suitably mnenomic (but short) name, then then pass it. Wouldn't it be nice if you could write the function right where you were going to pass it?

Of course the answer is "yes", otherwise we wouldn't have posed the question. The expression **(A => E)**, where **A** is an argument list (with the brackets omitted) and **E** is a sequence of expressions, is called a **lambda expression** (for historical reasons to do with Church's *lambda calculus*) or a *procedure literal.* Lambda-expressions are intended to be used for *short* procedures; otherwise they can make the code look cluttered.

Instead of defining **inc** above, we can write

```
(x => x + 1).mapCollection.println
```

for the same effect. To double each element, use **(x => x * 2)**; to square it, use **(x => x * x)**; to replace it with **0** use **(x => 0)**.

9.2.1 "full lexical scope"

Because we haven't said you can't, you may suppose that you can write lambda-expressions within procedures, and similarly, that those expressions may refer to, even assign to, local variables of those procedures. And you can.

```
define function const( x ) => (ignored => x) enddefine
```

const is a function that makes constant functions; **const(E)** is a function that delivers the value of **E** whatever it's applied to, so **const(0)** is a function [of one argument] that always delivers **0**. The variable **x** which is the parameter to **const** is available to the lambda-expression even after **const** has finished executing.

This behaviour (that the variables of a function are visible to lambda-expressions within that function, and that they live on after the function returns) is called *full lexical scoping*. It's relatively uncommon in programming languages (with the notable exceptions of Common Lisp, Scheme, Pop11, and Smalltalk), but it is your friend.

9.3 hole-expressions

Lots of little lambda-expressions are just an operator (or function call) with one (or two) arguments "missing"; like **inc** and its brethren above. For these Spice allows you to write a special form of lambda-expression, the *hole expression*, which is a function call or operator invocation with some of its arguments replaced by *holes*. The usual hole is written **?**, and it stands for "the argument"; it makes the application or invocation into a lambda expression.

So **? + 1** is another way of writing **(x => x + 1)**, where **x** has been replaced by the hole and the argument declaration is unnecessary.

? - 1 is the function that subtracts **1** from things, **42 - ?** is the function that subtracts things from **42**, and **? * ?** is the function that squares things. Both **?**'s refer to the same argument.

Just in case you ever want to use holes for functions with more than one argument, there are as many holes as you like, written ?**1**, ?**2**, and so on; **?** is shorthand for **?1**. A hole-expression has as many implied arguments as the biggest hole it uses; the Spice compiler will be curious about expressions with missing holes (eg **?19 + 1**).

9.4 combining functions

Because it's possible to write functions that take and deliver other functions, its possible to write *combining functions* who's job is just to manipulate other functions; this can result in a style known as *higher order programming*.

The standard function **Then** [note the capital T!] takes two functions **f** and **g** and returns a new function that takes some arguments **X**, applies **f** to them, and then applies **g** to the result; it might be written as:

```
define Then( f: Function, g: Function ): Function =>
```

```
    (args... => args.explode.f.g)
enddefine;
```

Note that **explode** generates arbitrarily many results, all of which become arguments to **f**; all of **f**s results become arguments to **g**; and all of **g**s results become the results of the lambda-expression. Nothing special has to be done by the programmer to keep track of how many arguments and results are returned.

Chapter 10

more on absent

We mentioned earlier that the value **absent** is the default value of type **Any**. It's used in Spice as the "missing" value, when no more constructive value is available, for example as the result of a failed lookup in a table.

It turns out that code like "the value of X, unless it's absent, in which case the value of Y" appears quite often. Spice allows this to be written compactly using the operator ||.

X || **Y** has the value **X**, unless it's **absent**, in which case it has the value **Y**; **Y** is not evaluated unless it has to be (*ie*, **X** is **absent**). It's rather like a pumped-up **or** where any non-absent value counts as **true**.

Similarly, **X && Y** is **absent** if **X** is **absent**, and the value of **Y** otherwise; again, **Y** is only evaluated if it has to be (*ie*, **X** isn't **absent**). It's rather like a pumped-up **and**.

Chapter 11

classes

So far we've only uses (some) of the built-in Spice types: integers, strings, arrays, and procedures. Spice also allows you to define your own data-types using *classes*.

11.1 class definition

A class definition allows you to define a new type for objects which you can use in your code. The simplest class definition introduces a type with named slots:

```
define class Pair
    slot front: Any := absent
    slot back: Any := absent
enddefine;
```

This defines a new class object, **Pair**, with two slots called **front** and **back**. The **slot** declaration is similar to a **var** declaration, except that it makes variables "inside" objects; you can leave out the initialiser and type in the usual way.

The name **Pair** becomes a new type name. You can use it as the type part of a variable declaration: **var p: Pair** means that **p** can hold only **Pair** objects. (The default value of **Pair** is a Pair with both slots **absent**.) Or it can be used to type a procedure argument.

You can print out the class if you wish; **Pair.println** will print something like **<Class Pair extends <absent>>**. (We say "something like" because it's possible to change the default way objects get printed.)

You can make new **Pair** objects with the expression **new Pair()**. Each such object has a **front** and a **back**, which start off as **absent**, and which you can

change. Every object also responds to the function **typeOf** by delivering its class object.

11.2 updaters

Suppose we've declared **var p := new Pair();** so that **p** holds a new **Pair** object. We can assign to the **front** and **back** slots of **p**:

```
p.front := 1, p.back := "two";
println( p.front, p.back );
```

prints **1 two**. In Spice, it's possible to call procedures on the left-hand-side of an assignment, in which case what happens is that the *updater* of that procedure is called. Just as Spice makes procedures to access the **slot**-slots of a class value, it gives them updaters to change those values[1].

You can define your own updaters if you wish:

```
define function x.foo := y as
    println( "updating", x, "with", y )
enddefine;
```

The **:= y** says this this is defining the updater of **foo** (and there had better be a definition for **foo** elsewhere; you can't define the updater of a procedure that doesn't exist) and that **y** is the name of the parameter which is the "right-hand-side of the assignment". **y** can be typed, as usual.

While an updater can do whatever it likes, it is usual that updaters are given only to access functions, and that they do the "obvious thing"; after an assignment **x.foo := E** then the expression **x.foo** should deliver the value **E** that was assigned.

If **foo** is a procedure, then **foo.updater** is the updater of that procedure, if it has one, and **absent** otherwise.

11.3 methods

A class can also define *methods* which act on objects of the type it defines.

```
define class Pair
```

[1] Unlike some languages (Common Lisp, Dylan), it is the procedure *value* that has the updater, not the procedure *name*. This means that you can call the updaters of procedures passed as parameters or stored into data structures. This is the bevahiour found in Pop11.

```
#
    slot front: Any := absent
    slot back: Any := absent
#
    define method wipe() =>
        this.front := absent, this.back := absent
    enddefine;
#
    define method setPair(x,y) =>
        this.front := x, this.back = y
    enddefine;
#
enddefine
```

This version of **Pair** defines two methods, **wipe** and **setPair**. **wipe** sets both slots to **absent**, and **setPair** sets them to the given parameter values. The name **this** is bound to an implicit additional parameter, which is the instance of **Pair** to be changed.

Methods are procedures, and you call them in the usual ways; if **p** is an instance of this **Pair**, then **p.wipe** will wipe it, and **p.setPair(1,2)** will set its **front** to **1** and its **back** to **2**. This reveals that **this** is bound to the *first* argument value. In fact, if it takes your fancy, you can use a different name than **this**, using the same function-definition syntax that we saw earlier:

```
define method self.wipe as
    self.front := absent; self.back := absent; none
enddefine
```

would be a suitable replacement for **Pair**'s version of **wipe**.

We'll see more about methods when we discuss inheritance, later.

11.4 initialisers

The **new Pair()** syntax is a bit limited, because you can't easily set new values for its slots. If you want to create a new object *and* set the values of its slots, you can define an *initialiser*. Here's yet another version of **Pair**:

```
define class Pair
    slot front: Any := absent
    slot back: Any := absent
    define init pair(x,y) =>
        this.front := x, this.back := y
```

```
    enddefine
enddefine
```

You can make a new initialised **Pair** by writing **new pair("hello","world")**. This works by making a new **Pair** and then handing that on as the **this** for **pair**. Any values returned by an initialiser are discarded; the result of a **new** expression is the newly constructed object.

You can have as many initialiser definitions as you like for a class. As a special favour, you can use **define init** (with no name) inside a definition of a class **P** to define overloadings of **new P(X)**.

11.5 inheritance

A class can *extend* an existing class. Here's one example; we might choose to make triples an extension of pairs.

```
define class Triple extends Pair
    slot side
    define init triple(x, y, z) extends pair(x, y) =>
        this.side := z
    enddefine;
endclass;
```

Triple has all the slots that **Pair** does, and one more: **side**. **front** and **back** automatically work on **Triple**s. **Triple**s *are* **Pair**s, and then some. The initialiser **triple** works by first invoking the initialiser **pair** in its **extends** clause to do the first two elements, and then assigning the third itself.

Not just **front** and **back**, but *any* function that works on **Pair**s and doesn't explicitly exclude non-**Pair**s will work on **Triple**s.

11.6 overloading functions

How does this happy state of affairs come about? When the argument type is the name of a class (like **Pair**), then it will accept values of any of its extensions (like **Triple**).

Sometimes, we want the behaviour of a method to be different in sub-classes than in the parent. In this case, the parent must define the method to be **generic** and mark the varying argument type with **:-**, and the different definitions for the sub-classes must be declared as **specific** and mark the same argument(s) with **:-**. This works outside as well as inside classes; the only difference is that inside a class, the implicit **this** argument is added to the method signature.

Here's an example of generic procedures outside a class:

```
define generic size( x:- Any ) as 0 enddefine;
define specific size( x:- Small ) as x @logBase 2 enddefine;
define specific size( x:- String ) as x.length enddefine;
```

This defines the **size** of an integer to be its bitswidth (and blows up if **x** is **0**), the **size** of a string to be its length, and the size of anything else to be **0**. The choice of which definition of **size** to use is made when **size** is called, by looking at the types of its arguments.

Chapter 12

type-expressions

So far we've said little about the type-expressions that can be written following **:**, except that certain built-in names (**Any**, **Small**, **String**) are allowed, and the names of any classes.

In fact types can be expressed with *expressions*, which have the same syntax and semantics as value-expressions. There are four important type-expressions in Spice: *multiple*, *lots*, *array*, and *non-optional* types.

12.1 multiple

If **T** and **U** are types, then (**T, U**) is the *multiple type* which is a value of type **T** then a value of type **U**.

12.2 lots

If **T** is a type, **T**** is the *lots of T* type, which is some unspecified number of **T**s as multiple values. The return type of a procedure defaults to being **Any****.

12.3 arrays

If **T** is a type, then **{T}** is the type *array of T* or *T row*. Note carefully that the Spice array constructor **{E}** *always* makes values of type **{Any}**, because Spice arrays can hold any kind of value; to make eg an **{Small}** you have to invoke a special constructor **IntArray(E)**.

12.4 optional types

Unless otherwise specified, **absent** is not permitted where a value of a specific (non-**Any**) type is expected; so a variable typed as **Small** cannot legally be given **absent** as its value. This can be over-ridden by using the **optional** type constructor; if **T** is a type, **T??** is the type which all the values of type **T** and also the value **absent**.

Chapter 13

packages

So far we've seen fragments of Spice code out of context for illustrative purposes. In real life, Spice code code should be organised into *packages*. A package is a collection of procedures, variables, and classes which work together to provide a coherent service to the programmer who uses that package.

A package starts with a *package header*, which identifies the package and the version of Spice it was intended for, specifies any other packages it may need, and then continues with the *package body*, which is a sequence of declarations and expressions.

The header starts with the *spice specification part*, which consists of the reserved word **spice**, a version string, and possibly some *preference settings*. The version string is a string specifying the version of Spice that the package is supposed to work with, and the preference settings control the Spice compiler.

Otherwise the header starts with the reserved word **package** and the package name, which is a series of simple names deparated by dots. For example,

```
spice "release.2.0";
package this.is.an.example;
```

Spice does not constrain package names to conform to file names on your local system[1], but it's a good idea to put a package whose name ends in **.foo** in a file called **foo** with a suffix (sometimes, misleadingly, called the "file type") acceptable to your Spice implementation, for example **.spi**[2].

[1] An IDE for Spice is *required* not to insist that package names form a path through a filing system hierarchy, and is *required* to accept package name components longer that the local filing system component names, and with "strange" characters in.

[2] The current Spice prototype compiler allows the mappings from package names to filestore names to be specified in package mapping entries in its initialisation file, as part of a proof-of-concept.

13.1 public and private

A package exists to make a service available to its users, which it does by making some of its identifiers visible to those users. By default, *none* of the identifiers in a package are visible. However, declarations can be marked as *public* by writing *declaration qualifiers*. To make a public variable, for example:

```
public var exposed: String := "hello"
```

declares a public string-only variable called **exposed**. You can mark a variable **private** instead, but since this is the default, it won't make much difference. **public** and **private** are keywords.

You can declare a function **public** in the same way:

```
define public inc( x ) => x + 1 enddefine;
```

More importantly, you can declare a **class** public. If you do so, the class name is public, *and so are all the non-slot methods declared in it.* The slot methods (ie the names of **slot**-variables) are still private by default; you can declare them public explicitly, and you can declare the other methods private explicitly.

13.2 imports

public is one side of the coin; **imports** is the other. A package can *import* another one in its header.

```
package using.example;
import this.is.an.example;
```

This makes all the public identifiers from **this.is.an.example** visible inside **using.example**. If you don't specify where this other package is to be found, the Spice compiler will use a set of (implementation-specific) rules to locate it, but you can be explicit:

```
import this.is.an.example from "/home/hedgehog/example.spi";
import this.is.an.example from
    "http://cdollin/Spice/Modules/example.spi";
```

Following the **from** is an expression (almost always a string literal) with the URL for the source of the package.

Spice does not require than imported packages are written in Spice; it is expected that implementations will define ways to import packages written in commonly-used languages such as C, C++, Java, and Javascript.

13.3 readonly

It's possible to qualify public variables and slots as *read only*, which means that they can be read but not changed by importing packages. For ordinary variables, all that need be done is to prohibit assignment to them; but for slots, which are accessed by procedures, it's not quite so simple.

When a slot is marked read-only

```
public [readonly] slot magic := 42;
```

(the square brackets mark **readonly** as a modifier, as it is not a reserved word) then the compiler makes *two* versions of **magic**; one for local use, and one for export. *Only the local version has an updater.* This can cause the interesting situation:

```
package readonly.example;
#
define class P
    public [readonly] slot magic = 72;
enddefine;
#
define function testEqual( x ) as
    println( x == magic )
enddefine;
#
endpackage
#
# now run testEqual from another package
#
package Q
import readonly.example;
#
testEqual( magic );
#
endpackage
```

which will print **false** as the exported **magic** is a different function to the local one.

13.4 protected access and aliases

It's possible to import a package, but to declare that you need to specify its name everywhere you specify its variables:

```
import protected readonly.example;
```

To refer to **magic** now you need to prefix it with the last part, the *leafname*, of its package name: **example::magic**. (You don't use the full name of a package except in **package** and **import** declarations.)

The main use of **protected** is to allow an imported package to define lots of useful names (usually constants) without polluting your own packages namespaces.

If you don't like the last part of the package name, or if it would be ambiguous, you can change the part you specify:

```
import protected boo = readonly.example;
```

Now **boo** is an alias for **readonly.example**, replacing its leafname, and we can write **boo::magic**.

13.5 pervasive import

A package that imports some identifiers – say, **brick** and **concrete** – from another package does not by default re-export them; they're not part of its *public interface*. They can be injected into the public interface by qualifying the import with **public**:

```
public import readonly.example;
```

Now this package exports **example::magic**.

Chapter 14

switches

The power of Spice's overloaded functions means that you won't write switches so often as you might in a more conventional language. But you still need to be able to choose from finite selections of integer and string values, so Spice has a general **switch** construct:

```
switch x
    case 1 then statement1();
    case 2 case 3 then statement2();
    case "hello" then # nothing
    case "world" then statement3();
    else statement4();
endswitch
```

If **x** has the value **1**, then **statement1()** is executed. If it has the value **2** or **3**, then **statement2()** is executed. If it has the value **"hello"**, nothing happens; if it has the value "world" **statement3()** is executed. And if it has none of these values, **statement4()** is executed.

The **switch** expression and the **case** expressions can be any Spice type; they're not restricted to numeric values. String and symbols are particularly useful. The **case** values need not even be compile-time constants, although that makes the switch less efficient, and in the interests of efficiency, if a switch has a compile-time case label of **N** then that takes priority over any run-time case label evaluating to **N**.

There's no need for a **break** or similar construct to leave the **switch**; the case body starts with its first statement and ends just before the next **case** or **else**. Only one **else** is allowed per switch.

If there's no **default**, it's as though **else none();** had been written.

Chapter 15

numeric literals and units

So far we've taken numbers pretty much for granted, writing them as digit sequences with the occasional _ for visual clarity. Spice has an extensive set of *numeric literals*.

All numeric literals start with a digit, and all numeric literals may contain underbars (which are ignored; they are there for the reader's visual convenience). When numeric literals contain letters, their case is irrelevant.

Numeric literals can be written in many bases: a non-decimal base is specified by writing it at the front of the number and following it with **x** (or **X**). The rest of the literal is interpreted as being expressed in that base, with letters being used for the extra digits; so the base cannot be more than 36. As a special case, **0x** can be used as in C and Java, for base 16.

A numeric literal may have a *scaling factor*, which is the letter **e** (or **E**) optionally followed by a sign ("+" or "-") and a series of digits. The digits represent a number in decimal, and the scaling factor is the literal's base raised to that power. If the literal's base is greater than **10** then the sign is not optional (otherwise **0x4e5** would be ambiguous). If present, the scaling factor is before the units.

A numeric literal that has a base less than or equal to **10** may end with a series of letters specifying the **units** in which the value is expressed. Typical units are **px** (pixels), **in** (inches), **s** (seconds), **em** (ems – a printing unit), and **mm** (millimetres).

The arithmetic operators work on values with units in the "natural" way, so you can add and subtract values with like units. (You can't just add a number to a unit value, though; **42in + 1** will generate a run-time error.) What's more, if the units are different but have the same underlying dimension (eg both lengths), then they can still be added and subtracted, by converting them to a common scale.

You can multiply values with units by numbers, in the obvious way. You can multiply values with units together, too; **6in * 7in** is **42 in in**, or 42 square inches. Division works in the corresponding inverse way.

An *integral* literal is a series of digits, possibly with a base specifier (in which case the digits can include letters), possibly with a scaling factor and units. A *floating* literal is a series of digits, a decimal point, and a second series of digits; again, it may include a scaling factor and units.

Chapter 16

fake XML syntax

To make simple manipulations of XML data possible, Spice has a built-in XML data type and an XML-like syntax for constructing those values.

```
val x = <snark> 99 </snark>;
```

will construct an XML node with tag the string **snark** and a single element, the value **99**, and initialise **x** with it. The value is an instance of the built-in class **XMLNode**, which has three useful methods: **xmlTag** will return the tag of the node, **xmlChildren** will return a sequence of the elements of the node, and **xmlAttributes** will return a sequence of the attributes.

A tag constructed by the <**tag**> syntax will be a literal string, but Spice allows it to be any value you like; just use as a tag an expression in brackets. In this (and other) cases it's a pain to have to repeat the tag in the </**tag**> part, so Spice allows the special closing tag </> which closes the most recent unclosed tag.

```
println( <(34 / 2)> 42 </> );
```

will print **17**. The elements between the opening and closing tags are expressions, not literal text (so Spice XML isn't real XML text, just a useful fake), and of course can include more XML constructs:

```
<chapter><verse>"hello"</>,1+2,<verse>"world"</></chapter>
```

Note the sneaky commas needed to separate elements.

Each attribute in the sequence returned by **xmlAttributes** is a **Maplet**, a single key-value pair; **maplet.mapName** is the key, and **maplet.mapValue** is the value. The attribute-setting syntax:

```
<tag attrOne=1 attrTwo="pi" attrThree=x attrFour=(Expr)> </>
```

allows arbitrary expressions to be bound to attributes, but for syntactic reasons if the expression isn't a constant or variable it has to be enclosed in brackets.

When working with XMLNodes, Spice multiple values are especially useful, because they can be constructed by loops:

```
<answer>
for x in someInterestingList do
    computeZeroOrMoreValuesFrom( x )
endfor
</answer>
```

All the answers from **compute...** become child elements of the **answer** node.

Chapter 17

enumerations

Sometimes you want a collection of distinct named values, for example as the names of options (**small**, **medium**, **large**, **jumbo**). Rather than using **val** integer variables (which can be confused with plain integer values) or strings/symbols (vulnerable to silent mis-spellings), Spice allows you to define *enumeration values*:

```
define enum Size = small, medium, large, jumbo enddefine;
```

Size becomes a new type name, and **small**, **medium**, **large**, and **jumbo** become new values of that type. **print(medium)** will print **medium**; the values retain their names. Each value also has an associated small integer, starting at **1** (for the first) and taking on successive values; you can see this number using **magnitude**, so **large.magnitude.println** will print **3**, and you can construct enum values from numbers: **new Size(4)** is identical to **jumbo**. Using values outside the range will generate an error.

Spice's enumerations are just shorthand. An **enum** type is a class extending the built-in **Enum** class, which has slots for the name and magnitude of its values. Each **enum** value is an instance of its class[1]with the slots set appropriately.

[1]Spice implementors are encouraged to implement **enum** values as efficiently as possible, taking advantage of their specialised nature.

Chapter 18

all sorts of unmentionable things

18.1 extended relational expressions

In Spice, the relational operators have a special syntactic property; you can write expressions like **x <= y <= z** to mean **(x <= y) and (y <= z)**. The expression **y** will be evaluated at most once. The operators do not have to be the same (**x = y <= z** is fine) and you can chain them up as long as you like (**x = y = z = h = q** is fine, too).

18.2 op

Sometimes you'd like to get at the functions that implement Spice's infix operators, usually so that you can pass them to higher-order procedures (eg for zipping two arrays together with a specified procedure such as addition). The form **op Operator**, where **Operator** is a Spice operator, gets a procedure that has the effect of that operator; so **op +** is a procedure of two arguments that adds them together and **op =** is a procedure of two arguments that tests them for structural equality.

op and and **op or** deliver ordinary procedures of two arguments (the special syntactic property that means that $\mathbf{E}_1$ **and** $\mathbf{E}_2$ need not evaluate $\mathbf{E}_2$ is lost) as does **op =**, **op <=**, etc (the chaining property is lost).

18.3 super and extends

Sometimes, in an overloaded procedure definition, you wish to invoke the more general version of that procedure; you do the general case, then you do your more specific one. The identifier **super** in a procedure is bound to the next most general definition of that procedure.

An initialiser (ie a **method new**) *must* invoke its more general case, and what's more, must do so before it starts its own initialisation. Rather than using **super** (which in any case doesn't have quite the right effect), that pre-initialisation is described in the **extends** clause of the initialiser, which provides calls to the other initialisers. If the extends clause is left out, it is as though you'd written **extends Parent()** where **Parent** is the parent class of this one.

18.4 shared slots

A class can have a shared slot, introduced by **shared** in place of **slot**. A shared slot is a single location accessible through any instance but shared by them all. (A shared slot is different from a variable declared in a class; it is accessed by method-call syntax, as ordinary slots are, rather than simply by its name, as variables are.)

Part III

the Spice reference manual

Chapter 19

introduction

19.1 a note on syntax

Syntax is described using an extended BNF. Terminal symbols that are names are in **bold**; terminals that are symbols are in bold and quoted, eg, "++". Non-terminals are in `this` font.

Grouping is shown by round brackets. Optional elements are enclosed in square brackets, eg in **"?"** [**Integer**] the **Integer** may be omitted. A suffix operator ***** is used to mean "zero or more of", while a postfix operator + means "one or more of".

The infix operator **E ** X** is used to mean "Zero or more **E**s separated by **X**s"; **X** is usually a comma or semicolon. Similarly infix **E ++ X** means "*one* or more **E**'s separated by **X**'s".

All the syntax definitions in the body of the document are collected together in appendix B.

Chapter 20

lexis

A Spice program is made up of a series of *tokens*. Tokens are separated by *token gaps*. Token gaps are unimportant except that they may serve to separate tokens that would otherwise combine into single tokens (eg the two tokens **x** and **then** must be separated by a gap, otherwise they would be recognised as a single token **xthen**).

A token gap is any sequence of spaces, newlines, horizontal or vertical tabs, and form-feeds, possibly containing comments (see below).

A token is a *word*, a *mark*, or a *literal*. A word is a sequence of letters and digits; a mark is a sequence of mark characters; and a literal is a string, character, numeric, or unit constant. The lexis makes no substantive difference between those words (or marks) that are reserved and those that may freely be used as identifiers (or operators).

20.1 words

A word is a series of letters, digits, or underbars, starting with a letter. Words are case sensitive and of unlimited length.

By convention, type names start with upper-case letters and non-type names do not, and capitalisation rather than underbars are used to mark word boundaries in names.

20.2 marks

A mark is either a *simple mark* or a *compound mark*. A *simple mark* is one of the characters "(", ")", "{", "}", "[", "]", ";", ",". A *compound mark* is a series

of mark characters, where a mark character is one of " !@%^&*-+=|:.?/<>".

"<>", "+++", "|->" etc are all possible Spice marks, even though they have no predefined meaning; they are reserved for future expansion.

Because of the use of "<" and ">" in the XML-like syntax, there are special rules for tokens starting with them, allowing the XML-like syntax to be a little more compact: leading sequences "</>", "</", "><" are broken down into their component single-character marks.

20.3 comments

A comment is either the character "#" followed by all characters up to a newline or end-of file, or the mark "/*" followed by any series of characters excluding "*/" and then by "*/".

20.4 literals

There are four kinds of literal in Spice: numeric literals, string literals, character literals, and regular-expression literals. The numeric literals may be decorated with units.

20.4.1 numeric and unit literals

A numeric literal represents an integer or floating-point number, possibly with units, possibly in a radix other than the default **10**. Within a numeric literal, underbar characters may be present; they are immediately discarded, being there only for presentation (eg to make **7_476_294** readable).

plain integer literals

A plain integer literal is a series of decimal digits. It represents the obvious number in radix **10**. There is no limit to the number of digits permitted, and the correct integer value is preserved; Spice will use values of the **BigNum** type if necessary.

radix-specified integer literals

A radix-specified integer literal consists of a radix specifier, which is a series of decimal digits followed by the letter **x** (in either case), followed by a series of letters and digits. The letters must be consistent with the radix, in that if the radix is **k** then no letter past **a + (k - 10)** is permitted; if the radix is **10** or

less, any letters terminate the numeric part of the literal. The special case **0x** can be used in place of **16x**.

integer literals with units

A plain integer literal, or a radix-specified integer literal with radix **10** or less, may be immediately followed by a unit name, which is a series of letters. The integer value represents a value in those units. Thus **45cm**, **19mile**, and **6seconds** are all legal, assuming that those units have been declared.

The value denoted by **nU**, where **n** is the number and **U** the units, is the measure value with **n** as its magnitude and the unit named by **U** as its unit.

plain floating literals

A floating literal is a series of decimal digits, a decimal point, and another series of decimal digits; it represents the "obvious" floating-point number. It may be followed by an exponent specifier, which is the letter **e**, optionally followed by a sign (+ or -), followed by a plain integer literal.

radix-specified floating literals

A radix-specified floating literal is a radix specifier followed by a series of letters and digits, a decimal point, a series of letters and digits, and an optional exponent specifier.

The radix specifier gives the radix of the floating point number but not the exponent. If the radix is **10** or less, then letters terminate the numeric part of the literal. If the radix is more than **10**, the sign in any exponent specifier is not optional.

floating literals with units

A floating literal with radix **10** or less may be followed by a unit name, as for integer literals with units.

20.4.2 string literals

A string literal is a character sequence starting and ending with the double-quote character **"**. The characters between the quotes are any characters except quotes (any flavour), newlines, control characters, or backslashes except as permitted by *escape sequences*. These escape sequences are shared with character literals and Spice-style regular expressions, see below.

- \n – a newline
- \f – a form feed
- \v – a vertical tab
- \r – a return
- \b – a backspace
- \t – a horizontal tab
- \s – a space
- \\ – a backslash
- \" – a double quote
- \' – a single quote
- \` – a reverse quote
- \&stuff; – an entity reference; **stuff** should be an allowable HTML entity reference, and the escape sequence denotes that character.
- \[, \], \{, \}, \|, *, \%, \? – these characters stand for themselves, even if they occur in an regular expression; they are said to be *protected*.
- \(Expression) – this is an *interpolation*, where the value of the expression is inserted into the string.

Any other backslash sequence **X** is reserved.

A string with an interpolation[1]is not a literal constant; it may evaluate to a different string value each time the literal is executed. The expression is evaluated and each of its multiple values converted to a collection of characters inserted into the string.

- character values are inserted as themselves
- string values are inserted as their component characters
- integer values are treated as the Unicode characters with the corresponding values

[1]Interpolation has not been tested in the current Spice implementations.

20.4.3 character literals

A character literal has the same syntax as a string literal, except that the delimiters are the single quote ' instead of the double quote.

A character literal denotes the multiple values formed by all its constituent characters. Thus ' ' denotes no values, **'x'** denotes the single character **x**, and **'spoo'** denotes the characters **s**, **p**, **o**, **o**.

20.4.4 regular expression literals

Spice has two regular expression syntaxes: a *traditional* syntax and a *native* syntax. The traditional syntax is based on existing regular-expression notations with little modification, and the native syntax is designed to mesh cleanly with the string syntax. Regular expressions are a late addition to Spice, and many details have not been settled, pending experimental implementation.

A traditional regular expression literal starts with the sequence // and then is a sequence of characters terminated by a closing /. The sequence cannot include any unprotected /. The exact permitted syntax of the sequence is under discussion, but we expect to conform as far as is possible with the syntax used by the Java regex package.

A native regular expression (NRE) starts with the sequence /" and then is a sequence of characters terminated by a closing "; the sequence cannot include an unprotected quotes. Within a native regular expression, all the usual string escapes apply with the same meaning, and protected characters stand for themselves and have no meta-meaning.

Our proposal is that the characters **?|*{}[]%** are special [that is, do not stand for themselves] within NREs. **%** is the *marker character* and the rest are meta-characters.

- *character classes*, [XYZ]: the square brackets enclose compact character-class expressions.
- *grouping*, {E}: braces are grouping brackets which have no effect on back-references.
- *repetition*, E*: suffix * is the Kleene star.
- *wildcard*, ?: matches any character.
- *alternation*, E|F: matches either E or F.

Other syntactic jobs are performed by sequences introduced by the marker character. They are expected to be less frequent and so a larger overhead to use them is acceptable.

Chapter 21

top-level syntax

A Spice program is composed of a collection of *packages*. A package consists of a *header*, which identifies the package and what it relies on, and its *body*, which is a series of *definitions* possibly interspersed with *executions*. Definitions define variables, procedures, classes, enumerations, and units; executions are expressions that perform some run-time activity.

21.1 programs

def1. Program ::=

```
[(Spice)] (Package* | PackageBody)
```

def2. Spice ::=

```
'spice' String (',' (Name ':'  Expr))*
```

The **String** identifies the version of Spice that the writer assumed when the package was written; it is a dot-separated sequences of integers or names (eg "1.0" or "5.beta"). The **Name-Expr** pairs define settings for compiler preferences; each **Expr** must be evaluable at compile-time.

The **spice** preamble is intended for future expansion.

21.2 modifiers

Modifiers are annotations to declarations. They allow a declaration to be accompanied by arbitrary information. Modifiers can appear in two different positions, *marked* modifiers and *unmarked* modifiers; marked modifiers can decorate declarations not introduced by **define** (eg, **var** and **slot**), and thus need a more restricted syntax.

def3. OpenModifier ::=

```
MarkedModifier Modifier*
```

def4. Modifier ::=

```
MarkedModifier
| '[' Name [(MarkedArgument)]** ',' ']'
```

def5. MarkedModifier ::=

```
'public'
| 'private'
| 'protected'
```

def6. MarkedArgument ::=

```
Literal
| '(' Expression ')'
```

Modifiers are discussed in the sections that require them, except we shall mention here some common important modifiers@

- **public**: the identifier(s) are put into the public interface of this package, ie, they can be used in packages that import this one.
- **private**: the identifier(s) are *not* put into the public interface of this package; they cannot be referred to outside it at all. **private** is the default, except for identifiers declared in **public** packages, as below.
- **protected**: protected identifiers are public but can only be referred to in other packages using the **alias::name** notation; see also **protected import**.
- **facet(Names)**: puts the declared identifier into the specified facets instead of **public**.

21.3 packages

def7. Package ::=

```
MarkedModifier 'package' PackageName Facets PackageBody
```

def8. PackageName ::=

```
Word++ '.'
```

def9. Facets ::=

```
['facet' Name++ ',']
```

A package name is a sequence of dot-separated identifiers. The last name in the sequence is the *leafname* of the package; apart from in **package** and **import** constructs, packages are referred to only by their leafname or an alias for it.

Top-level identifiers declared in a **public package** are public by default. Top-level identifiers declared in a **private package** are private by default. If the package is not declared **public** or **private**, it defaults to **private**.

Each package has several *facets*, which are named external interfaces; they are declared by the optional **facet** part of the package declaration. Every package has at least two facets, **public** and **private**, but **private** cannot normally be seen by importing packages.

def10. PackageBody ::=

```
Import** ';' Bundle** ';'
```

def11. Bundle ::=

```
Definition+ Expr*
```

A package body consists of its import declarations followed by its own definitions. All the names exported by the packages which have been imported are available in all the bundles of the package. Each bundle is a sequence of definitions, possibly followed by some expressions; each **Bundle** is as long as possible. The names declared by a **Bundle** are *visible throughout that bundle*; declarations need not lexically preceed their use.

Expressions are evaluated when the package is loaded.

def12. Definition ::=

```
ProcedureDef
| VarDef
| ClassDef
| UnitDef
| EnumDef
```

A definition is a procedure, class, unit, enumeration, or variable definition.

21.4 imports

def13. Import ::=

```
MarkedModifier
'import'
OpenModifier
[(Alias '=')]
PackageName
[('facet' (Name** ','))]
[('from' Expr)]
```

def14. Alias ::=

```
Name
```

An **import** definition identifies a package to be imported and possibly a location to import it from; that location is given as a String-valued expression

representing a URL. The name can be given directly or by the evaluation of an expression.

If the modifier **public** is given, then all the identifiers in the imported package's public facet are added to this package's public interface. This allows packages to act as "collectors" for definitions from several other places.

An imported identifier **x** can be referred to as **leafname::x**, where **leafname** is the leafname of the package (if **Alias=** is omitted) or the *alias* **name** (if **Alias=** is given). Unless the modifier **protected** is given, it may also be referred to as **x**.

If two (or more) packages with the same leafname are imported, at least one of them must be given an alias. If two (or more) packages export the same identifier **x**, then **x** must be referred to using the **alias::x** notation.

If **facet** is present, only the names in the specified facets are imported.

21.5 procedure definitions

The full power of a procedure definition is available through **define** options, which allow generics, methods, plain functions, and initialisers to be declared.

def15. ProcedureDef ::=

```
FullProcedureDef
| CompactDef
```

def16. FullProcedureDef ::=

```
'define'
Modifier*
('method' | 'function' | 'generic' | 'init' | 'specific')
[(':')]
Header
[('returns' Type)]
[('extends' CommaExpr)]
[('super' Header)]
ProcedureBody
'enddefine'
```

def17. CompactDef ::=

```
'def' Header '=>' Expr
```

A procedure declared by **def** is monomorphic; it has a single definition and cannot be redeclared.

A procedure declared by **generic** is a polymorphic procedure; the **generic** definition gives its most general argument types and its default body.

A procedure declared by **specific** is a dynamic overloading of the named generic

procedure (which must exist and be visible).

If a **generic**, **specific**, or **method** definition appears within a class, it is given an implicit first argument called **this** whose type is that class, unless the **plain** modifier has been supplied. Outside a class, **plain** does nothing. **function** and **def** never supply a **this**.

A procedure declared with **init** is an *initialiser*: it is used to fill in the fields of freshly-constructed objects.

init procedures may only be declared within classes. A name used for a **init** procedure cannot be used as the name of an ordinary procedure, nor for **init** procedures in a different (visiable) class.

The **Header** defines the procedure argument's names and types, the procedure name itself, and the "expected shape" of calls to the procedure. **returns** gives the type of the value(s) returned by a procedure; if omitted, a return type is inferred from the body of the procedure.

def18. Header ::=

```
CallShape
| Arg '->' CallShape
| CallShape ':=' Arg
```

The allowed **Modifier**s include **public**, **protected**, and **private**.

If the updater argument is given, this definition defines the *updater* of the named procedure, which must be declared in this package and have a compatible **CallShape**.

def19. CallShape ::=

```
PrefixShape
| DottedShape
| InfixShape
```

def20. PrefixShape ::=

```
NameA Arglist
```

def21. DottedShape ::=

```
ArgB . NameA [(Arglist)]
```

def22. InfixShape ::=

```
ArgA @ NameA [(ArgB)]
```

There are three different **CallShape**s; the conventional prefix form **f(x)**, the method-oriented form **x.f(a)**, and the infix-oriented form **x @f y**. **NameA** is the name of the procedure being defined in all cases. The arguments are given by the optional **Arglist** for the prefix form, by the **Arglist** and **NameB** for the method form, and by **ArgA** and **ArgB** for the infix form; **ArgA** and **ArgB** are both **Arg**s, see below.

A **method** is implicitly in method form, with **this.** assumed as the first argu-

ment unless overridden explicitly. For **init** it is permitted to omit the **NameA**; in this case, the initialiser is that called by expressions of the form **new C** when **C** is the name of the enclosing class.

def23. Arglist ::=
```
'(' Args ')'
```

def24. Args ::=
```
Arg** ','
```

def25. Arg ::=
```
Name
| Name ':'  Type
| Name ':-' Type
| Name '==' Expr
| Name '...'
```

An **Arglist** is just a comma-separated list of **Arg**s. An **Arg** is a **Name**, optionally followed by a **Type** introduced by **:** or **:-**, or optionally followed by a value introduced by "=="; or by **...**, making it an *indefinite* argument, bound to an array of all the extra arguments to the procedure. The arguments are declared as local variables[1]of the procedure. At most one indefinite argument is allowed, but it may appear anywhere in the argument list.

If the **:Type** form is used, the argument takes part in polymorphic dispatch. Otherwise it does not.

== Expr is equivalent to **: T** where **T** is a type whose only value is the value of **Expr**, which is evaluated at compile-time.

def26. ProcedureBody ::=
```
StatementSeq
| ArgList '=>' ProcedureBody
| Arg '=>' ProcedureBody
```

A procedure body is introduced by => or **as** and ends with **enddefine**. If the body is empty, the arrow may be omitted. If the body has the form **A=>B**, the procedure is *curried*; the body is itself a function expression.

21.5.1 multiple definition

A **generic** procedure may have many different definitions, with the rest being given as **specific**s or **method**s. The different definitions may have different numbers of arguments (up to any indefinite arguments) and no two definitions can have the same sequence of types for their arguments.

Furthermore, for any pair of definitions with signatures (the type-sequence of

[1]All argument identifiers are immutable, ie, cannot be assigned to, but if they are data-structures, their components can be changed.

its arguments) $\mathbf{T}_1$ and $\mathbf{T}_2$, there must be a definition with type-sequence $\mathbf{T}_3$ where the **i**th element of $\mathbf{T}_3$ is a super-type of the **i**th elements of $\mathbf{T}_1$ and $\mathbf{T}_2$ (note that $\mathbf{T}_3$ might be equal to one of $\mathbf{T}_1$ or $\mathbf{T}_2$).

A call of such a polymorphic method will execute the most specific instance of it, the one whose formal arguments are as specific as possible while matching the actual arguments. A procedure with **K** arguments does not match a call with **N** /== **K** arguments, unless it has an indefinite argument and **N**>**K**.

In particular, methods can be redeclared in classes, when they over-ride the definitions[2]in their parents.

21.5.2 super and extends

An overloaded procedure may wish to invoke the "next most general" definition to complete (or start) its work. The identifier **super** is bound to that next most general definition. When there is ambiguity about that, it is resolved by the **super** clause[3]of the function, which takes the form of a function call where the operands are all type names; **super** is bound to the version of the function that takes those types as its argument.

To avoid ambiguities, any Spice generic function, when called, must have enough definitions that this is never ambiguous.

An initialiser in a class **C** with parent class(es) **A (B...)** is required to invoke initialisers of all of those classes. This is done in the **extends** clause[4]of the initialiser; this clause is a comma-separated list of initialiser calls, where each called procedure must be an initialiser of a parent class and each parent class must have at least one initialiser called.

21.6 variable definitions

A variable declaration introduces one or more identifiers, may give their initial value, and may give their type.

def27. VarDef ::=

```
MarkedModifier
('val' | 'var')
NameDecl
[(':=' Expr)]
```

[2]If a method has definitions of different arities, all definitions with indefinite arguments are, for the purposes of dispatch, extended to the same length by adding additional penultimate arguments of type **Any**.

[3]This use of **super** has not been tested in the fire of the current Spice implementation.

[4]Similarly, this use of **extends** has not been tried in practice.

def28. NameDecl ::=

```
(OneDecl | '(' OneDecl++ ',' ')')
```

def29. OneDecl ::=

```
Name [(':'  Type | '...')]
```

The **Modifier** may include **public**, **protected**, and **private**. A **var** may have modifier **readonly**, in which case it cannot be updated outside of its defining package; **val**s can never be assigned to. All the names declared in a single **val** or **var** declaration share the same **Modifier**.

The **NameDecl** may initialise one or more names to the value(s) of an **Expr**; if the **Expr** is omitted, they are initialised to the default value of their type; if no type is specified, they are given type **Any**.

A **NameDecl** will usually contain a single **OneDecl**, but it may contain several, in which case the **Expr** must be present and they are initialised from it as for assignment. The **...** form is permitted for at most one **OneDecl**; it bundles all the (remaining) values from the **Expr** into an array which is assigned to the **Name**.

It is forbidden for **OneDecl**s to be typed if the entire **NameDecl** is typed, and *vice versa*.

When a **var** or **val** appears at top-level, or within a class, it defines a permanent variable that comes into existance when the package is loaded and goes away only when no references to it remain.

When a **var** or **val** appears within a procedure, it defines a local variable; a new location is allocated each time the declaration is executed. This location does not share with any other location. It persists as long as there are any references to it, which means that if it is not captured by any lambda-expression or hole-expression referring to it it may be disposed of when its scope is left (if not sooner).

21.7 class definitions

A Class defines a shape shared by a collection of objects which are its instances. Every Class is itself an object, and has a particular instance called its *prototypical instance*.

def30. ClassDef ::=

```
'define'
Modifier
'class'
Name
[('extends' CommaExpr)]
ClassElement**
SEMI
'enddefine'
```

The **Name** is defined to be a constant bound to the class object, and is suitable for use as a type name.

A class may be qualified as **public**, **protected**, **private**, **array**[5], and **abstract**. If it is qualified **array**, its instances are array-type objects, and can be indexed with integers. A **public** class has its **method**s and **function**s (but not its **slot**s) declared **public** by default. A class qualified **abstract** cannot have direct instances; trying to **new** it will fail.

If the **extends** clause is present, its **CommaExpr** must be a comma-separated sequence of names of other classes[6], and this class inherits all of their slots. (Slots that are inherited twice along different routes of course only count once in the new class.)

def31. ClassElement ::=

```
Definition
| Statement
| SlotDecl
```

The elements of a class are normal definitions and expressions, and its slots. Definitions within a class acts exactly like definitions outside a class – they declare top-level permanent identifiers. Within a class, method definitions by default give their implicit **this** argument that class as their type.

def32. SlotDecl ::=

```
('slot' | 'shared')
Name
[(':'  Type)]
[(':=' Expr)]
[('implements' Name2)]
```

Slots are declared by **slot** and **shared** declarations. Each **slot** declaration arranges that objects of this type have a slot in them, suitable for storing values of the given **Type**, and defines a procedure **Name1** of one argument which has the class as its type and whose action is to deliver the value of its slot; that procedure also has an updater which alters the value of that slot.

If **implements Name2** is specified, then **Name2** must be a generic procedure,

[5]The **array** modifier is not implemented by the current Spice compiler.
[6]The current Spice compiler only implements single inheritance.

and the slot declaration provides an overloading of that procedure for arguments of the current class.

A **shared**[7] declaration does not allocate any slots in the object, but makes a procedure which accesses (and updates) a single shared location.

If a slot or shared identifier is qualified **readonly**[8], then the procedure that it exports has no updater, although the locally-visible version does.

When an object of this type is constructed, the **Expr**s of its **slot**s are run to get the initial values for those slots.

A class definition for **Spoo** allows the expression **E is Spoo** as a test for the value **E** being an instance of **Spoo** (or a subtype).

[7] Not implemented in the current Spice compiler.
[8] Not implemented in the current Spice compiler.

Chapter 22

small syntax

Most of the rest of the syntax of Spice is in terms of *expressions*. Spice expressions denote tuples of values; we say thay are *multiple valued*. Spice expressions may be evaluated in two modes; *value* mode, when they are evaluated for their value and *update* mode, when they are evaluated to update some location or data structure.

Unless otherwise specified, an expression is illegal in update mode. All function calls and operator applications are legal there; every procedure **P** can have an updater **U**, which is the procedure that is called when a call of **P** appears as the target of an assignment.

22.1 expressions

A **Statement** is an expression. The value of a sequence of **Statement**s is the multiple value produced by concatenating the multiple values of its consitiuent statements.

def33. Statement ::=
```
CommaExpr++ ',' SEMI
```

def34. StatementSeq ::=
```
Statement*
```

def35. Expr ::=

```
Name
| Hole
| Literal
| Expr '...'
| PreOp Expr
| Expr PostOp
| Expr InOp Expr
| Expr AssignOp Expr
| '(' Expr ')'
| Expr '.'  DotExpr [(Expr)]
| Expr '@' DotExpr [(Expr)]
| Expr '[' Expr ']'
| LambdaExpr
| 'new' Name [('(' Expr ')')]
| 'once' Expr
| '{' CommaExpr '}'
| QuoteExpr
| LikeXMLExpr
| IfExpr
| RepeatExpr
| ForExpr
| SwitchExpr
```

Operator expressions are disambiguated by the "usual" precedence rules. Here, **Expr** is an expression that cannot contain top-level commas.

def36. CommaExpr ::=

```
Expr++ ','
```

22.1.1 name

A **Name** is either a simple identifier, or an imported identifier **short::id**, where **short** is the short name of a package and **id** is the name of the identifier within it.

22.1.2 hole

A Hole is a "virtual operand" used when making partial applications. Thus the expression **? + 1** is a function that adds 1 to things. The expression **?2 - ?1** is reverse subtract; the arguments to a partial application are numbered **1..N**, left-to-right.

def37. Hole ::=

```
'?'  [(Integer)]
```

In an expression with several hole operands (to the same operation), either all

the holes are tagged, or none are (and they are all the same argument).

The notion of "partial applications" above is generalised for the special syntactic operators **and**, **or**, ||, and **&&**.

22.1.3 literal

Literals may be symbols, strings, characters, numbers, or reserved literals. Symbols, strings, numbers, and reserved literals are themselves lexical items.

def38. Literal ::=
```
StringLiteral
| NumberLiteral
| CharacterLiteral
```

22.1.4 explosions

The value of the expression **E...** is the *explosion* of the value of **E**, ie all the components of all the component of **E**.

In update mode (ie as the target of an assignment) **E** must be an updatable expression; the value assigned to it is an array constructed of all the source values.

22.1.5 prefix operators

The value of a prefix expression is found by applying the procedure bound to the prefix operator to the value delivered by the operand, *except* for the special operator **once**, which arranges that its operand is evaluated at most once.

22.1.6 postfix operators

The value of a postfix expression is found by applying the procedure bound to the postfix operator to the value delivered by the operand.

def39. PostOp ::=
```
LexicalPostfixOperator
```

[Note that @ can serve as a infix or postfix call marker.]

22.1.7 infix operators

def40. InOp ::=
```
LexicalInfixOperator
```

The logical operators **and** and **or**, and the relational operators ==, <=, >=, <, >, **!=**, /==, ==, /== are special in that they must always appear with left and right operands that each evaluate to exactly one value.

Further, the relational operators are *continued relationals*; if **R** and **S** are relational operators, the expression **a R b S c** is shorthand for **a R b and b S c**, except that **b** is only evaluated once.

The special infix operators **and**, **or**, ||, and **&&** are special: they only evaluate their right operand if necessary.

22.1.8 assignment operators

The assignment operator **:=** evaluates its left operand (the *target*) in update mode with its right operand being the *source*; all the source values must be consumed by the target.

The assignment operator "->" evaluates its *right* operand (the target) in update mode, with the source being its *left* operand; any excess values are retained as the value of the assignment expression.

The assignment operators **=:** and **<-** have the same meaning as those of their reversed forms, with the roles of left and right operand exchanged.

The source delivers some sequence of values; the target is a sequence of updateable expressions. If the sequence is a single procedure call, its updater is invoked and all the values (**:=** and **=:**) are, or a single value (**->**, **<-**) is, passed to it. Otherwise, At most one target is allowed to be a procedure call with an updater which takes an indefinite number of assigned values. (Otherwise, in **(f(), g()) = (1,2,3)**, where **f** and **g** both have indefinite updaters, there would be no way to tell how the values were to be distributed between **f** and **g**.)

Assignments are done right-to-left; each assignment target takes the appropriate number of values from the right-hand end of the value sequence and the rest of the value sequence is handed to the remaining targets. If the assignment target is an identifier, one value is assigned into it. If it is a procedure call, its updater is invoked, passing it the appropriate number of values.

22.1.9 bracketed expressions

The expression **(E)** has the same value that **E** does.

22.1.10 call expressions

The expression **x.f** is a call of **f** with argument(s) **x**, as are the expressions **f(x)** and **x @f**. The expression **x.f(y)** is a call of **f** with arguments **(x,y)**, as is **x @f**

y.

The arguments are evaluated left to right; the function is evaluated either before or after the operands. This document does not specify which.

When a call appears as the target of an assignment, eg **f(x)=E**, it is equivalent to the expression **(f.updater)(x)(E)**[1]; that is, the updater function specialises itself on the target arguments [allowing it to be overloaded on those arguments] and delivers a function which consumes the assigned values [and which may be separately overloaded on those values].

def41. DotExpr ::=

```
Name
| '(' Expr ')'
| 'new'
```

The expression permitted after a dot or **@** may be a **Name**, an arbitrary expression in brackets, or the reserved word **new** (which allows it to be used infix).

22.1.11 sequence expressions

The expression **{E}** is a *sequence expression*; it creates a new sequence by evaluating its operand and piling all the results into the new sequence.

22.1.12 index expressions

The value of the expression **a**[**i**] is that obtained by calling the *index function* associated with the type of **a**, supplying it with **a** and **i**. If **a** is an array, this attempts to index the array. If **a** is a table, it looks **i** up in that table.

When an index expression is used in update mode, the updater of the associated index function is called.

22.1.13 lambda expressions

A lambda-expression is an inline procedure definition; it is the procedure taking the specified arguments and computing the desired result.

def42. LambdaExpr ::=

```
'(' Args '=>' StatementSeq ')'
| 'fun' Args '=>' StatementSeq 'endfun'
```

[1] The current Spice implementation uses an earlier definition that has a similar effect in simple cases, but is inadequate to deal with overloaded functions: **(f.updater)(x,E)**.

def43. LambdaExpr ::=

```
'(' LambdaBody ')'
| 'fun' LambdaBody 'endfun'
```

def44. LambdaBody ::=

```
Args '=>' LambdaBody
| StatementSeq
```

22.1.14 quotation expressions

A quotation expression allows **Symbol** values, and sequences of **Symbol** values, to be denoted by an expression.

def45. QuoteExpr ::=

```
'`' QuotedItems '`'
```

def46. QuotedItems ::=

```
QuotedItem*
```

def47. QuotedItem ::=

```
Word
| '{' QuotedItems '}'
| '(' Expr ')'
| '^' Word
| '^^' Word
| '\\' Word
| [('\\')] Literal
```

A **QuotedItem** that is a literal or a word denotes the corresponding value; words are represented as Spice **Symbol**s (interned strings with a type of their own). An item which is an expression in brackets evaluates to the value of that expression; the form ^**Word** is equivalent to **(Word)**, with **Word** stripped of any special syntactic properties, and the form ^^**Word** is equivalent to **(Word...)**. A literal or word prefixed by \ stands for itself.

The form **{Q}** denotes the sequence formed out of the values denoted by **Q**. Thus quotations can express nested sequence structures which are mostly constant.

22.1.15 XML-like expressions

An XML-like expression is a constructor for trees (instances of the built-in class **XMLNode**) with a syntax modelled on that of XML, modified to fit into a programming language.

def48. LikeXMLExpr ::=

```
LeafyTree
| BranchingTree
```

def49. LeafyTree ::=

```
'<' TreeHead '/' '>'
```

def50. BranchingTree ::=

```
'<'
TreeHead
'>'
StatementSeq
'<'
'/'
[(TreeHead)]
'>'
```

An XML-like expression **BranchingTree** is a sequence of statements enclosed by fat brackets; tree heads wrapped in "<"-">" pairs. The closing bracket starts "</". As a special case, if the statement sequence is empty, the two fat brackets can be collapsed together to form a **LeafyTree**.

def51. TreeHead ::=

```
TreeLabel TreeAttribute*
```

def52. TreeLabel ::=

```
Literal
| Id
| Id ':'  Id
| Id '::'  Id
| '(' Expr ')'
```

The label of a tree can be a literal or a plain name interpreted as the string with the same spelling. Otherwise the label is dynamically evaluated. The form **Id:Id** is a special expression, meaning the value of the left-hand identifier concatenated with the spelling of the right. The remaining two forms have the same meaning as in plain expressions.

def53. TreeAttribute ::=

```
Name [('=' Expr)]
```

Each tree attribute is a binding of a name to a value. If the value expression is omitted, the value **true** is used. The name is not a variable; it means a string with the same spelling.

Evaluating an XML-like expression constructs an instance of **XMLNode** with three access methods: **xmlTag**, which delivers the tree label; **xmlChildren**, which delivers the sequence of values computed by the **StatementSeq**; and **xmlAttributes**, which delivers the sequence of attributes, each attribute being represented by a value of type **Maplet**.

Standard procedures (not described here) render XMLNodes into output streams compatible with the usual XML stream syntax.

22.1.16 new expressions

The expresssion **new X(A)** is a construction expression; **X** is a constructor value, and **A** the arguments to the construction. If the argument **(A)** is omitted, it does *not* mean that **X** is invoked on no values; instead the constructor *function* is delivered, as though one had written

```
(A... => new X(A...))
```

If **X** is a class value, then a new default-initialised instance of that class is constructed, and handed to the class's anonymous constructor. If **X** is a named constructor, that is, defined as **new X** in some class **C**, a new empty instance of **C** is constructed and passed to **X** with arguments **A**.

22.1.17 if

An **if** expression expresses a choice. Both verbose and compact if-endif forms are supported; in the short form, **if** - **endif** is replaced by **(** - **)**, "|" is used in place of **then** and **else**, and "|:" in place of **elseif**.

An **if** must have a matching **endif**, and an **unless** must have a matching **endunless**. **then** and **do** are equivalent in **if** and **unless** expressions. The **unless** form inverts the initial test.

def54. IfExpr ::=
```
LongIfExpr
| ShortIfExpr
```

def55. ShortIfExpr ::=
```
'('
Expr
'|'
CommaExpr
('|:'  Expr '|' CommaExpr)*
['|' CommaExpr)]
')'
```

def56. LongIfExpr ::=
```
('if' | 'unless')
Expr
Then
StatementSeq
('elseif' Expr Then StatementSeq)*
['else' StatementSeq)]
('endif' | 'endunless')
```

def57. Then ::=

```
'then'
| 'do'
```

22.1.18 repeat

A **repeat** loop repeats statements while (or until) a condition is true. On each iteratation, **Seq1** is executed, and then **Expr1** is tested. If it is satisfied (**true** for **while**, **false**for **until**) then **Seq2** is executed and the loop repeats. Otherwise, the loop terminates and delivers all the values computed by the loop body.

def58. RepeatExpr ::=

```
'repeat'
StatementSeq1
[(RepeatTest [('do' StatementSeq2)])]
'endrepeat'
```

def59. RepeatTest ::=

```
('while' | 'until') Expr1 [('then' Expr2)]
```

22.1.19 for

A **for** loop causes a body of code to be repeatedly executed while various conditions are **true** (or **false**), and variables are stepped "in parallel" along a sequence of values. When the loop terminates "naturally" (that is, by exhaustion of one of the sequences) then a termination clause is executed.

def60. ForExpr ::=

```
'for'
ForControls
[('do' StatementSeq)]
[('finally' StatementSeq)]
'endfor'
```

The controls are separated by **;**, although that's optional if the next control is a **while** or an **until**.

def61. ForControls ::=

```
ForControl
| ForControls ';' ForControl
| ForControls ForCondition
```

A **Control** is a binding of a variable to values, or an early-exit condition.

def62. ForControl ::=

```
ForBinding
| ForCondition
```

If the early-exit condition is satisfied (**while false** or **until true**), the loop terminates. If the **then** (for **until**) or **else** (for **while**) is present, its code is executed before the loop exits; otherwise the loop's **finally** code is executed.

def63. ForCondition ::=
```
'while' Expr 'else' Expr
| 'until' Expr 'then' Expr
```

A **ForBinding** binds names to sequences of values.

def64. ForBinding ::=
```
ForName 'in' Expr
| ForName 'on' Expr
| ForName 'from' Expr
| ForName 'to' Expr
| ForName 'from' Expr ('to' | 'downto') Expr
```

def65. ForName ::=
```
Name [(':'  Type)]
```

For-loop identifiers are declared there, are immutable, and are local to the loop. What is more, a *new* identifier is bound each time round the loop (this is noticable only if the loop body forms closures using lambda-expressions or holes which involve the identifiers).

The meaning of a **ForBindings** is that each **Name** introduced is given successive values from its **Expr**, in parallel (ie on the **N**th iteration each **Name** has its **N**th value), until at least one of the **Expr**s is exhausted. The notion of "successive values" is defined by the run-time type of the expression's value. It is not defined what happens if that value is modified during the execution of the loop body.

The **in** form takes all the values from the range of a collection type (eg all the characters from a string, all the elements of an array). The **on** form binds the identifier to the maplets of a collection, so that for an array or string the maplets are of the form **index ==> element**.

The form **from** $\mathbf{E}_1$ **to** $\mathbf{E}_2$ runs the control variable from the value $\mathbf{E}_1$ to the value $\mathbf{E}_2$. If **to E** is omitted, then the loop is unbounded (it will terminate only via **break** or if a parallel iteration terminates). If **from E** is omitted, the loop starts at **1**. If **downto** is used instead of **to**, the loop counts down rather than up.

22.1.20 switch

def66. SwitchExpr ::=
```
'switch' Expr (SCase | SElse) 'endswitch'
```

def67. SCase ::=
```
('case' Expr)+ 'then' StatementSeq
```

def68. SElse ::=

```
'else' StatementSeq
```

Each **case** clause starts with some number of **case** expressions (or **else**) and is followed by the expressions to evaluate when the switch expression takes the value of one of those labels.

A Spice switch may switch on *any* values; it is not restricted to integer constants (although it may be much more efficient on them). In particular, a Spice switch may switch on strings and symbols.

A **switch** can have at most one **else** clause.

The result from a **switch** is the result from the selected **case** or **else** statement sequence. Thus a **switch** may deliver multiple values.

22.1.21 var and procedure expressions

A **VarDef** and a **ProcedureDef** counts as expressions, but they deliver no values.

Note that this means that procedure definitions may be nested. A procedure may access and update the local variables of the procedure that it appears in; Spice has full lexical scope.

22.2 type expressions

A **type expression** is an expression. Its semantics is the same as that of ordinary expressions, but they must deliver values of type **Type**.

def69. Type ::=

```
Expr
```

The language of type expressions is intended to allow the programmer to give the compiler useful information about the program.

The basic type expression is the *name*[2]. A name used as a type expression should be the name of one of the built-in types or a class. **Any** is the name of the universal type, **Object** is the name of all object types.

If **T** is a type-expression, then **T??** represents the *optional type* of **T**; it is the type of values which may be **T** or may be the value **absent**.

If **T** and **U** are type-expressions, so is **T,U**, which represents the type of multiple values with first component(s) or type **T** and second component(s) of type **U**.

If **T** is a type expression then so is **T****, the type of lots of multiple values all of type **T**.

[2]Only names are legal type-expressions in this edition of the Spice compiler.

If **T** is a type-expression, **{T}** ("array of T", "row of T") is the type of arrays who's elements are required to be of type **T**.

Chapter 23

implemented but dubious features

The current Spice implementation includes exception handling in the style of C++ and Java using **throw** and **try-catch-endtry** expressions. We have our doubts about this form of handling, so we have relegated them to this part of the manual.

23.1 throw and try expressions

The expression **throw E** raises an exception with value the value of **E**. If this exception is not caught within the program, it is caught by the top-level Spice environment, which will deal with it in an implementation-specific fashion.

throw may appear as a **DotExpr**, so **E.throw** has the same effect as **throw E**.

The **try** expression allows exceptions to be caught and handled.

def70. TryCatchExpr ::=
```
'try' StatementSeq CatchSeq 'endtry'
```

def71. CatchSeq ::=
```
('catch' Arglist ['as')] StatementSeq)**
```

The **StatementSeq** of the **try** is evaluated and the **try** expression resturns its result if no exception is thrown. If an exception **E** is thrown, then the **catch** body who's **Arglist** best matches the thrown value is executed (just as though the **catch** blocks were alternative definitions for an overloaded function) and the reult of the **try** is the result of that **catch** block.

If there is no **catch (x: Any)** catch clause, it is as though the clause **catch**

(x:Any) with throw x had been supplied.

Note that this is *not* sequential testing. Sequential testing means that you have to write the general case last, not first, and makes things more order-dependant than they need be. Making the **catch** clauses share the semantics of procedure call makes the language more coherent and offers alternative implementation tactics.

Chapter 24

designed but unimplemented features

Some features of Spice have been designed but not implemente din the currenct compilers: unit and enum definitions.

24.1 unit definitions

A **unit** definition[1]informs the compiler about units and their dimensions.

def72. UnitDef ::=

```
'define'
OpenModifier
('unit' | 'units')
UnitDef++
','
'enddefine'
```

def73. UnitDef ::=

```
Word [('=' Expr)]
```

The **Word** of a **UnitDef** is declared to be legitimate unit names. If the **Expr** of a **UnitDef** is present, it must be an compile-time expression delivering an anonymous **Unit** value, which is named by the **Word**.

for example, we may have **define unit mile = inch * 12 * 3 * 1760 enddefine**, which defines **mile** in terms of **inch**, or **define unit length = new Unit(1,'length') enddefine**, which defines a new unit of dimension 'length'.

[1]Not implemented by the current Spice compiler.

24.2 enumeration definitions

def74. EnumDef ::=

```
'define'
OpenModifier
'enum'
Name
'='
Name++
','
'enddefine'
```

An **enum** declaration for **Foo** is shorthand for declaring a new class called **Foo** extending **Enum**, which provides two private slots (for the name and the magnitude of enumeration values) and an instance of this class for each of the **Name**s. The enumeration values start at **1** and the default value for **Foo** is an instance with value **0** and name **nullFoo**.

Implementors are encouraged to find efficient representations for enumeration values.

Chapter 25

scope rules

The scope rules for Spice define where identifiers are visible. They are intended to be natural and in most cases are straightforward.

A sequence of forms (eg, a package body, or a non-package) consists of sequences of consecutive declarations, *bundles*, interspersed with non-definition expressions. All the **Definition**s in a bundle are mutually in scope. In consequence, there is rarely any need for "forward" declarations. Non-definitions terminate this mutual scope to allow the rules for interactive execution to be the same as those for batch compilation.

classes do not introduce new scopes; the declarations within a class put names into the top-level namespace. The same name cannot legally declared more than once. (A **method** definition does not re-declare a name; it attaches additional definitions to an existing generic procedure.) Class slots must be uniquely named in a scope.

A procedure definition introduces a new scope; the arguments and the body share that scope. (Thus is it illegal to have two arguments with the same name, or an argument to have the same name as a top-level local of the procedure.) The body of the procedure is bundled in the same way as a package.

A **LambdaExpr** introduces a new scope. All the names from the surrounding scope are visible inside the lambda, with the same rights; in particular, it is permitted to assign to locals of the surrounding context, and for this to "work": Spice has full lexical scoping.

A **try** expression introduces a new scope which extends to the first **catch** clause (or to the **endtry** if there are no explicit **catch** clauses). A **catch** clause introduces a new scope which ends at the next [non-nested!] **catch** or **endtry**.

Each of the arms of an **if** or **unless** expression have their own scopes. The body of a **repeat** loop is a scope. (What's more, variables declared in the body of a

loop are re-declared *each time round the loop*.)

Each **SpiceCase** introduces a new scope which ends at the end of its **StatementSeq**.

A **for-do** statement introduces a new scope. The identifiers declared in a **ForBinding** are in scope in any succeeding **while** or **until** clause, and in the loop body, but *not* in the **Expr** parts of other **ForBinding**s of this loop. As for **while** and **until** loops, these identifiers are re-declared each time round the loop.

The right operand of an **and** or **or** operator has its own scope. (This odd rule is because **x and ((var y := 42), z)** is a legal expression, but then questions would arise about **y**'s existence or value if **x** were **false**.)

Chapter 26

the standard library

Spice has a large library of standard values and procedures.

26.1 generic procedures

- **x.typeOf**: the type of **x**.
- **x.printOn(s)**: send a human-readable representation of x down the stream s. Characters are sent as-is, integers are printed as their signed decimal representation [note: there ought to be a way of defining a negative numeric literal], strings are printed as their sequence of characters, arrays are printed as **{elem1 elem2 ...}** and objects are printed as defined by their overloading of printOn (and Class.printOn(s) does something sensible).
- **x.print** shorthand for x.printOn(standardOutput)
- **x.report**: shorthand for x.printOn(standardReport)
- **println(x...)**: **print** all the arguments in turn, prefixing all but the first with a space, and printing a newline after printing all the arguments.
- **x.copy**: make a copy of x. If x is atomic (Small, Symbol, Char, Procedure) just delivers x. If x is composite object, defaults to making a shallow copy of x. Overloadable for new types.
- **x.toString**: convert **x** to a string in the same way that **print** does.
- **equals(x, y)** the procedure implementing =. Overloadable on new types, but the programmer has to ensure that the new definition respects the usual rules for equality (reflexive, transitive, symmetric).

- If x and y are both Small, Symbol, Char, or Procedure, equals delivers x == y (ie identity). If x == y, equals delivers true.
- If x and y are both objects, by default equals does a slot-by- slot = test on all the init-slots that x and y have in common. If any of these fail, equals delivers false; otherwise it delivers true.
- If x and y are both String, equals does the obvious string equality test. If they are both arrays, equals does the obvious sequence-equality test.
- **x.hashCode**: a type-specific hashcode for the value **x**.
- **apply(f, x...)**: called when evaluating f(X) and f is not a function. Overloadable. On arrays and strings is equivalent to indexing. On tables is table lookup. Has the obvious updater. Allows objects to represent functions compactly. Is not implemented.
- **x.length**: overloadable. Should deliver the "length" of the object. On strings and arrays has the obvious meaning. On objects delivers the number of "obvious" slots (see explode). On atomic objects delivers 0.
- **x @asType t**: **t** must be a type. The value **x** is converted to that type in a type-specific way (**asType** is overloadable), or an exception is thrown. **x @asType String** is equivalent to **toString**.

26.2 numbers

Spice has several kinds of number[1]: small integers (**Small**), big integers (**BigInt**), short and long floating point values (**Float**, **Double**), ratios (ie fractions) (**Ratio**), and complex numbers (**Complex**).

The operators +, **-**, *****, /, **div**, **rem**, **%** work more-or-less as you'd expect on plain numbers (they work on some other values, too, for which see the section on **units**). **div** is integer division; the operands must be integers and the result is truncated toward **0**. **rem** is integer remainder; the operands must be integers and the result is the appropriate remainder. / will produce a floating-point value if the result is not an exact integer.

The operator **/:** is *exact division*; its result uses ratios to express non-floating non-integer components. This is the primary way (the only way, at root) that ratio values are generated.

The relational operators <=, <, >=, >, ==, /== work as you'd expect. Note that it's unwise to use == and /== on floating-point values.

There are also a number of standard functions.

[1] But only **Small** and **Double** are implemented.

- **x.abs** and its synonym **x.magnitude** is the absolute value of **x**; if **x** is not complex, its sign is made positive; if **x** is complex, it is its magnitude.
- **x.neg** and its equivalent **x.negative** invert the sign of the value of **x**. **x.positive** returns **x** if it is a (non-complex) number.
- **x @min y**, **x @max y** deliver the minimum (maximum) of their argument values **x** and **y**.
- **x @logToBase y** delivers the logarithm of **x** to the (positive integer) base **y**.
- **x @toPower y** delivers **x** raised to the power **y**.
- **x.sqrt** is the square root of **x**. An exception is thrown if the result must be **Complex** but this implemention does not support it.
- **x.round**, **x.floor**, **x.ceiling** convert the value **x** to an integer (details to be added).
- **x.cos**, **x.sin**, **x.tan** are the usual trigonometric functions. The argument is in radians.
- **x.inRadians** takes an angle in degrees **x** and converts it to radians. **x.inDegrees** takes an angle in radians and converts it to degrees.
- **x.acos**, **x.asin**, **x.atan** are the usual inverse trigonometric functions. Note that **atan** is only suitable for simple programs.

[various complex functions etc to be done]

26.3 enumerations

Enumeration values can be queried for their numeric value and name, and can be created from numeric values or names. Note that the number of enumeration values is fixed by the declaration; the constructor delivers existing values, not fresh ones.

- **e.index**: if **e** is an enumeration value, its index is its position in the list in which it was declared.
- **new E(n)**: if **E** is an enumeration type, the **n**th enumeration value that it was declared with, or an error is thrown if **n** is out of range.
- **e.enumName**: if **e** is an enumeration value, the symbol which names it.
- **new E(s)**: if **E** is an enumeration type, and **s** is a string or symbol, the enumeration value of **E** named by that string or symbol, or an error is thrown.

26.4 values with units

Spice has values with **units**. Values with units represent lengths, or times, or other inter-related quantities. All the unit procedures are imported **protected** from **spice.lang.units**.

A **unit** expresses an amount along a given dimension. A **basic dimension** is represented by a symbol naming that dimension. A **compound dimension** is the result of a product or quotient of dimensions.

- **Unit**: the type of units.
- **Unit(scale, unitName, dimension)**: make a new unit object. **dimension** must be a symbol or string; it specifies the dimension of the unit. (Typical values are **length**, **time**, **colour**.) **unitName** must be a symbol or string; it names the unit itself. [It is an error if a unit with that name already exists.] The **scale** expresses the scale of this unit in terms of the "canonical unit" for this kind; a scale of **1** defines the canonical unit.
- **U @unitProduct V**: **U** and **V** must be unit objects. The result is the product of those units.
- **U @unitQuotient V**: **U** and **V** must be unit objects. The result is the quotient of those units.
- **U.scale**: **U** must be a **Unit** object; the result is its scale.
- **U.name**: **U** must be a **Unit**object; the result it its unit name.
- **U.dimension**: **U** must be a **Unit** object; the result is its dimension.
- **x @inUnits y**: **x** must be a numeric value, and **y** must be a unit name (symbol or string) or a **Unit** object (eg from **unitNamed**). The result is a unit value with **x** as its number and the unit (named by) **y** as its unit.
- **n.unitNamed**: the **Unit** value named by **n**, which must be a string or a symbol.
- **u.number**: the number of the value **u**, which must be a value with units.
- **u.unit**: the unit of the value **u**, which must be a value with units.
- **u.explode**: **u** must be a value with units; exploding it gets the number and unit, in that order.

The arithmetic operators also work on values with units. There is a minor complication in that different units of the same kind may be inter-converted. Multiplication and division also work on **Unit** objects themselves to produce new **Unit** objects; this is intended for use in **define unit** definitions.

- **u + v, u - v**: **u** and **v** must be values with the same dimension. The number with the larger unit is scaled to the size of the smaller, and the result is the sum (difference) of the numbers, with the smaller unit as its unit. [*ie*, adding inches to miles gets a result in inches.]
- **u * v**: if **u** or **v** has units, the result has number **u.number * v.number** and units **u.unit @unitProduct v.unit**.
- **U * V**: if **U** and **V** are units or numbers, the result is a new unit which has dimensions the product of the dimensions of **U** and **V**, and number the product of the numbers of **U** and **V**.
- **u / v**: if **u** or **v** has units, the result has number **u.number / v.number** and units **u.unit @unitQuotient v.unit**.
- **U / V**: if **U** and **V** are units or numbers, the result is a new unit with dimensions the quotient of the dimensions of **U** and **V**, and number the quotient of the numbers of **U** and **V**.
- **u < v**: equivalent to **number(v - u) < 0**, and similarly for the other relational operators.

26.5 strings

Strings are sequences of Unicode characters in canonical form. Strings are immutable. Strings, like arrays, can be indexed using the **s[i]** notation. The infix operator ++ concatenates strings (and arrays).

There are a few built-in functions on strings.

- **s.length** is the length in characters of **s**.
- **s @indexOf x** delivers the index in **s** of the first occurance of **x**, which can be a character or a string. If **s** does not contain any occurence of **x**, **indexOf** delivers **absent**. The result has the same base as the string does.
- **s @lastIndexOf x** is the same as **indexOf**, except that it searches for the last occurence of **x**, not the first.
- **s @chopAt x**, where **x** is a string or character, delivers *two* results **(before, after)** such that **s == (before ++ x ++ after)** and **x** does not occur earlier in **s**.
- **s @chopAt x**, where **x** is an integer, delivers **(after, before)** such that **s = before ++ after** and **after.length == x** - s.stringBase.
- **s @split x** delivers an array of strings obtained by repeatedly chopping **s** with **x**.

- **s.toLower** is a copy of **s** with all upper-case characters replaced by their lower-case counterparts; **s.toUpper** is a copy of **s** with all lower-case characters replaced by their upper-case counterparts.

26.6 symbols

The *symbol literal* **'foo'** represents a *symbol* (a value of type **Symbol**[2]) whose *spelling* is the string **foo**. The important difference between symbols and strings is that there is only one symbol with a given spelling. (You can tell the difference using the Spice *identity operator* ==.)

You can index symbols using **s**[**i**], but if you do so, you are probably using symbols for something other than their intended use; supplying the programmer with an infinite set of mnenomic values.

- **s.intern**, where **s** is a string, delivers the symbol whose spelling is **s**, making it if necessary.
- **s.spelling**, where **s** is a symbol, delivers the string which is the spelling of **s**.
- **s.explode**, where **s** is a symbol, delivers all the characters of **s**, rightmost last.

26.7 booleans

The built-in operators **not**, **and**, **or** on booleans operate as you might expect; **not** negates its operand, **and** is boolean **and** and **or** is boolean **or**, neither of which evaluate their second operand unless it's necessary.

26.8 arrays

Array elements can be accessed and updated using the **a**[**i**] notation.

- **a.length**: delivers the length of the array **a**.
- **a @reduceBy (x, p)**: **a** must be an array, **x** some value, and **p** a dyadic procedure. The result is obtained by starting with **x** and repeatedly replacing it by the result of **p(result,ai)** for **ai** being bound to all the elements of **a** in turn.

[2]Symbols are not currently implemented.

- **a @join s, a.join**: equivalent to **a @reduceBy (s, catStringly)**; with one argument, **s** is taken to be the null string "".
- **a @sortInPlaceWith f**: sorts the array **a** according to the comparison function **f**. **f** should accept two arguments **(x, y)** and deliver **true** if **x** should go before **y** in the ordering and **false** otherwise. **sortInPlaceWith** is guaranteed to be stable if **f** is sane.
- **a @sortWith f**: makes a copy of **a** and sorts that in place with **f**, so far as anyone can tell. (The implementor is at liberty to make a more efficient version.)
- **a.reverse**: a copy of **a** with the elements reversed.
- **a.reverseInPlace**: reverses the order of the elements of **a**.

26.9 bits

Spice does not have standard operators for bit operations on integer values; instead, it has standard functions which you call in infix form. This reflects the designer's belief that bit operations are simply not used enough to warrent using up useful symbols for them, at least in *their* code.

- **x@bitNot**: the bitwise complement of **x** and **y**.
- **x @bitAnd y**: the bitwise **and** of **x** and **y**.
- **x @bitOr y**: the bitwise **or** of **x** and **y**.
- **x @bitXor y**: the bitwise **exclusive or** of **x** and **y**.
- **x @bitClear y**: **x @bitAnd y.bitNot**.
- **x @bitShl y**: **x** shifted left **y** places.
- **x @bitShr y**: **x** shifted right **y** places. Note: the sign of **x** is *always preserved* for plain (**Small** or **Big**) integer values.

26.10 procedures

There are several standard functions on procedures.

- **p.updater**: if **p** is a procedure, then delivers its updater (or **absent** if if hasn't got one).

- **p @apply** x: **p** must be a procedure, and **x** a collection of values; applies **p** to all the values in **x**, ie, is **x.explode.p** except that **x** must be a compound type.

The different kinds of procedures can be recognised by predicates.

- **x.isProcedure** is **true** when **x** is any kind of procedure, and **false** otherwise.
- **x.isMethod** is **true** if **x** is defined by a **method** definition, and **false** otherwise;
- **x.isFunction** if **x** is defined by a **function** definition, and **false** otherwise;
- **x.isConstructor** is **true** when **x** has been defined with **method new**, and **false** otherwise;
- **x.isLambda** is **true** when **x** is the result of a lambda-expression or hole-expression, and **false** otherwise.

26.11 dictionaries and tables

The type **Lookup** is the parent type of a variety of mapping data types; the *simple* mapping types map single values to single values, while the *compond* mapping types map tuples of values to tuples of values. There are no direct instances of **Lookup**. The type **Maplet** is the type of pairs of (simple) values. The type **Dictionary** is an extension of **Lookup** that maps **Symbol**s to values.

- **l.length**: the number of key-value associations stored in the lookup **l**.
- **l.explode**: the values stored in the lookup **l**.
- **l.maplets**: the maplets of the lookip **l**.
- **p @mapOver l**: apply the procedure **p** to each of the key-value pairs in **l** in turn. **p** will be called with two arguments, being the key and the value. *All the results of* **p** *will be returned.*
- **l @hasKey k**: if **l** is an instance of an extension of **Lookup**, **hasKey** returns **true** if **k** is a key for a non-default value in **l** and **false** otherwise.
- **l @fetch k**: if **l** is an instance of an extension of **Lookup**, **fetch** gets the value bound to the key **k**. If no value is bound, some type-specific computation is performed. **fetch** has an updater which alters the bound value, or creates a binding if none already exists.
- **l @apply k**: if **l** is an instance of an extension of **Lookup**, applying it runs **fetch**.

- **m.key**: the key part of **m**, ie, the value which would be looked up.
- **m.value**: the value part of **m**, ie, the value that would be returned when looking up the **key** part.
- **Dictionary()**: constructs an empty **Dictionary** object with default value **absent**. A **Dictionary** is an extension of **Lookup**.
- **Dictionary(d)**: constructs an empty **Dictionary** with default value **d**.
- **Dictionary(d, k_1, v_1, ..., k_n, v_n)**: constructs a **Dictionary** with default value **d** and bindings which bind k_i to v_i. If several k_i are equal, the last one wins. All the k_i must be **Symbol**s.
- **Table**: an extension of **Lookup** with constructors in the same style as **Dictionary**. A **Table** can have key arguments of any type; they are compared using == (and hashed using **hashCode**, qv).
- **FatTable**: an compound mapping extension of **Lookup**. **FatTable** has similar constructors to **Dictionary**, except that the keys and values must all be arrays; all the keys must have the same length, and all the values must have the same length. A **FatTable** maps tuples of values to tuples of values.
- **f @fetch (k_1, ..., k_n)**: if **f** is a **FatTable**, look up the value(s) specified by the sequence of keys k_i, which must be the same length as the key arrays used in the constructor of **f**. Deliver the associated multiple values. The updater of **fetch** will store multiple values into **f**.
- **Maplet**: a type expressing a single binding of a key to a value.
- **k ==> v**: a Maplet mapping **k** to **v**.

26.12 input and output

Spice includes some simple I/O operations in its core; the type **Pathname**, the type **File**, and the type **Stream**.

- **s.parsePath**: **s** must be a string or symbol. It is parsed into a **Pathname**, or an exception thrown if it is illegal in some way. (See below for the syntax of pathnames).
- **p.pathScheme**, **p.pathRoot**, **p.pathDirs**, **p.pathName**, **p.pathSuffix**, and **p.pathType**: the corresponding components of the pathname. If the component was omitted in the originating string, the result is **absent**.
- **p @openIn**: **p** must be a pathname (or string or symbol, which is immediatley parsed into a pathname). The external entity named by the path is opened for input. **openIn** delivers a **File** object.

- **p @openOut**: **p** must be a pathname (or string or symbol, which is immediatley parsed into a pathname). The external entity named by the path is opened for output. **openOut** delivers a **File** object.
- **f.openedOn**: if **f** is a file, then the pathname it was opened on.
- **f.close**: **f** must be a file; if it is not closed, it is hereby closed, committing all writes (if open for output).
- **f @readInto (b, w, l)**: **f** must be a file open for input. **readInto** reads the next **l** bytes (or fewer) into the buffer **b**, which must be a byte array, starting at position **w**. An exception is thrown if **w** or **w+l** would be outside the bounds of **b**. The number of bytes actually read is returned as the result.
- **f @writeFrom (b, w, l)**: **f** must be a file open for output. **l** bytes from the byte array **b** are written, starting at offset **w**. An exception is thrown if **w** or **w+l** would be outside the bounds of **b**.
- **f.inFrom**: **f** is a file open for input, or a pathname (which is immediately **openIn**'ed) or a string (treated as a pathname). The result is an byte **InStream** on the file.
- **f.outTo**: **f** is a file open for output, or a pathname (which is immediately **openOut**'ed), or a string (treated as a pathname). The result is an byte **OutStream** on the file.
- **s.next**: **s** must be an **InStream**; the result is the next object from **s**, or **absent** if the file has been exhausted. **next** has an updater which puts objects back onto the stream. Arbitrarily many objects can be put back, and they need not correspond to objects originally present in the stream.
- **s @out x**: **s** must be an **OutStream** and **x** an object of the appropriate type; that object is appended to the stream.
- **s.streamFile**: if **s** is a stream based on a file, delivers that file; otherwise an exception is thrown.

26.12.1 pathname syntax and accessors

A pathname has several components. The basic elements are **Word**s which, within this section, are sequences of characters not otherwise reserved to the pathname syntax.

In this section, the identifier **p** is presumed to contain a **Pathname** value.

def75. Pathname ::=

```
[(Scheme)]
[(Root)]
[(Directories)]
[(Name)]
[(Suffix)]
[(Type)]
```

def76. Scheme ::=

```
Word ':'
```

The **Scheme** of a pathname directs how the components are to be interpreted. The standard schemes are **file**, **http**, **ftp**, and **socket**.

p.pathScheme is a **Symbol**.

def77. Root ::=

```
'//' Dotted
```

The **Root** identifies where the pathname is anchored. For an **http** or **net** pathname, the **Root** is the IP address where the named entity is located. For a **file** scheme, the **Unit** identifies some root in the filing system in an implementation-specific way.

p.pathRoot is an array of **Symbol** values, one for each word-part of the **Dotted**.

def78. Directories ::=

```
[('/')] (Dotted '/')++
```

The **Directories** identify some place within the **Root** where an entity is found, by giving a sequence of **Dotted**s. For a **file** scheme, thet are the names of filing-system directories. For an **http** scheme, they are successive components of the directory part of the URL. For a **socket** scheme, they identify the port number to be used for the socket; typically there is but one **Dotted** and it names an IP service.

p.pathDirectories is an array of **Symbol**s, each symbol being the spelling of a **Dotted**.

def79. Name ::=

```
Word
```

The **Name** is the leafname of the entity being described.

p.pathName is a Symbol.

def80. Suffix ::=

```
'.'  Word
```

The **Suffix** is the suffix part of the entity name.

p.pathSuffix is a **Symbol**.

def81. Type ::=

```
';' Word
```

The **Type** is the type of the entity. When a pathname is being used to create an object, is specifies the type of object to create. When it is being used to access an existing object, the object should be of a compatible type.

p.pathType is a **Symbol**.

def82. Dotted ::=

```
Word++ '.'
```

26.13 types

The standard types are

- **Lookup**, the parent type of data dictionaries.
- **Maplet**, a helper type for data dictionaries.
- **Table**, the type of tables.
- **FatTable**, the type of compound tables.
- **Small**, the type of small integers.
- **Number**, the type of all numbers – integers, floats, rationals (when implemented), complexes (ditto).
- **Complex**, the type of complex values.
- **Ratio**, the type of ratios.
- **BigInt**, the type of big integers.
- **Float**, the type of all floating-point numbers.
- **String**, the type of strings.
- **Procedure**, the type of all procedures.
- **Bool**, the type of booleans.
- **Char**, the type of characters.
- **Any**, the type of anything whatsoever.
- **Object**, the type of all objects (ie things defined by classes).
- **Date**, the type of dates.
- **Unit**, the type of units.
- **Pathname**, the type of pathname objects used to represent URLs (and local file names, etc).

Chapter 27

glossary

- **absent**. The "missing" value.
- **Any**. The universal type; all values are compatible with **Any**.
- **class**. A value representing a template for a collection of other values called its **instances**.
- **explode**. (a) the name of a procedure which explodes its single argument. (b) to take a value and deliver all of its consituent values.
- **enum**. An abbreviated **class** definition which describes a new class and all of its values.
- **hole**. The marker **?** in an expression, representing a parameter position for an implicit lambda-expression wrapped round the smallest enclosing application.
- **initialiser**. A procedure declared with **define init** which initialises its first argument (usually called **this**) according to the values of its remaining instances.
- **lambda expression**. An expression describing a procedure by giving its arguments and body.
- **method**. A **procedure** expected to be called using dot-notation and defined within a **class**. Methods have a usually-implicit first argument usually called **this**.
- **measure**. A value consisting of a number and a unit, meaning that many of that unit.
- **multiple values**. A Spice expression can evaluate to zero or more values, and how many values is determined at run-time, not compile-time. Thus a

function call, or a loop, may deliver several results, which may be assigned or embedded into a larger expression.

- **new**. Syntactic form for building instances of a class.
- **package**. A named collection of definitions and executable expressions; the unit of encapsulation.
- **predicate**. A **procedure** of one argument that returns a single **Bool** value.
- **procedure**. A piece of code invoked from elsewhere to perform some specified task. A procedure takes some number of arguments, performs some action, and delivers some number of results.
- **slot**. (a) A named location within (instances of) a class. (b) the procedure which gives access to such a location.
- **super**. Used in method calls to invoke the "next more general" method.
- **unit**. The value used in a **measure** to specify the units of that measure, eg inches, seconds, x's, and so on.
- **updater**. The part of a **procedure** that implements its behaviour when it is called as the target of an assignment.

Chapter 28

syntactic summary

This section gathers together all the syntax from the rest of the document.

def1. Program ::=
```
[(Spice)] (Package* | PackageBody)
```

def2. Spice ::=
```
'spice' String (',' (Name ':'  Expr))*
```

def3. OpenModifier ::=
```
MarkedModifier Modifier*
```

def4. Modifier ::=
```
MarkedModifier
| '[' Name [(MarkedArgument)]** ',' ']'
```

def5. MarkedModifier ::=
```
'public'
| 'private'
| 'protected'
```

def6. MarkedArgument ::=
```
Literal
| '(' Expression ')'
```

def7. Package ::=
```
MarkedModifier 'package' PackageName Facets PackageBody
```

def8. PackageName ::=
```
Word++ '.'
```

def9. Facets ::=
```
['facet' Name++ ',']
```

def10. PackageBody ::=

```
Import** ';' Bundle** ';'
```

def11. Bundle ::=

```
Definition+ Expr*
```

def12. Definition ::=

```
ProcedureDef
| VarDef
| ClassDef
| UnitDef
| EnumDef
```

def13. Import ::=

```
 MarkedModifier
 'import'
 OpenModifier
 [(Alias '=')]
 PackageName
 ['facet' (Name** ',')]
 ['from' Expr)]
```

def14. Alias ::=

```
Name
```

def15. ProcedureDef ::=

```
FullProcedureDef
| CompactDef
```

def16. FullProcedureDef ::=

```
 'define'
 Modifier*
 ('method' | 'function' | 'generic' | 'init' | 'specific')
 [(':')]
 Header
 ['returns' Type)]
 ['extends' CommaExpr)]
 ['super' Header)]
 ProcedureBody
 'enddefine'
```

def17. CompactDef ::=

```
'def' Header '=>' Expr
```

def18. Header ::=

```
CallShape
| Arg '->' CallShape
| CallShape ':=' Arg
```

def19. CallShape ::=

```
PrefixShape
| DottedShape
| InfixShape
```

def20. PrefixShape ::=

```
NameA Arglist
```

def21. DottedShape ::=

```
ArgB . NameA [(Arglist)]
```

def22. InfixShape ::=

```
ArgA @ NameA [(ArgB)]
```

def23. Arglist ::=

```
'(' Args ')'
```

def24. Args ::=

```
Arg** ','
```

def25. Arg ::=

```
Name
| Name ':'  Type
| Name ':-' Type
| Name '==' Expr
| Name '...'
```

def26. ProcedureBody ::=

```
StatementSeq
| ArgList '=>' ProcedureBody
| Arg '=>' ProcedureBody
```

def27. VarDef ::=

```
  MarkedModifier
  ('val' | 'var')
  NameDecl
  [(':=' Expr)]
```

def28. NameDecl ::=

```
(OneDecl | '(' OneDecl++ ',' ')')
```

def29. OneDecl ::=

```
Name [(':'  Type | '...')]
```

def30. ClassDef ::=

```
'define'
Modifier
'class'
Name
[('extends' CommaExpr)]
ClassElement**
SEMI
'enddefine'
```

def31. ClassElement ::=

```
Definition
| Statement
| SlotDecl
```

def32. SlotDecl ::=

```
('slot' | 'shared')
Name
[(':'  Type)]
[(':=' Expr)]
[('implements' Name2)]
```

def33. Statement ::=

```
CommaExpr++ ',' SEMI
```

def34. StatementSeq ::=

```
Statement*
```

def35. Expr ::=

```
Name
| Hole
| Literal
| Expr '...'
| PreOp Expr
| Expr PostOp
| Expr InOp Expr
| Expr AssignOp Expr
| '(' Expr ')'
| Expr '.'  DotExpr [(Expr)]
| Expr '@' DotExpr [(Expr)]
| Expr '[' Expr ']'
| LambdaExpr
| 'new' Name [('(' Expr ')')]
| 'once' Expr
| '{' CommaExpr '}'
| QuoteExpr
| LikeXMLExpr
| IfExpr
| RepeatExpr
| ForExpr
| SwitchExpr
```

def36. CommaExpr ::=

```
Expr++ ','
```

def37. Hole ::=

```
'?'  [(Integer)]
```

def38. Literal ::=

```
StringLiteral
| NumberLiteral
| CharacterLiteral
```

def39. PostOp ::=

```
LexicalPostfixOperator
```

def40. InOp ::=

```
LexicalInfixOperator
```

def41. DotExpr ::=

```
Name
| '(' Expr ')'
| 'new'
```

def42. LambdaExpr ::=

```
'(' Args '=>' StatementSeq ')'
| 'fun' Args '=>' StatementSeq 'endfun'
```

def43. LambdaExpr ::=

```
'(' LambdaBody ')'
| 'fun' LambdaBody 'endfun'
```

def44. LambdaBody ::=

```
Args '=>' LambdaBody
| StatementSeq
```

def45. QuoteExpr ::=

```
'`' QuotedItems '`'
```

def46. QuotedItems ::=

```
QuotedItem*
```

def47. QuotedItem ::=

```
Word
| '{' QuotedItems '}'
| '(' Expr ')'
| '^' Word
| '^^' Word
| '\\' Word
| [('\\')] Literal
```

def48. LikeXMLExpr ::=

```
LeafyTree
| BranchingTree
```

def49. LeafyTree ::=

```
'<' TreeHead '/' '>'
```

def50. BranchingTree ::=

```
'<'
TreeHead
'>'
StatementSeq
'<'
'/'
[(TreeHead)]
'>'
```

def51. TreeHead ::=

```
TreeLabel TreeAttribute*
```

def52. TreeLabel ::=

```
Literal
| Id
| Id ':'  Id
| Id '::'  Id
| '(' Expr ')'
```

def53. TreeAttribute ::=

```
Name [('=' Expr)]
```

def54. IfExpr ::=

```
LongIfExpr
| ShortIfExpr
```

def55. ShortIfExpr ::=

```
'('
Expr
'|'
CommaExpr
('|:'  Expr '|' CommaExpr)*
['|' CommaExpr)]
')'
```

def56. LongIfExpr ::=

```
('if' | 'unless')
Expr
Then
StatementSeq
('elseif' Expr Then StatementSeq)*
['else' StatementSeq)]
('endif' | 'endunless')
```

def57. Then ::=

```
'then'
| 'do'
```

def58. RepeatExpr ::=

```
'repeat'
StatementSeq1
[(RepeatTest ['do' StatementSeq2)])]
'endrepeat'
```

def59. RepeatTest ::=

```
('while' | 'until') Expr1 ['then' Expr2)]
```

def60. ForExpr ::=

```
'for'
ForControls
['do' StatementSeq)]
['finally' StatementSeq)]
'endfor'
```

def61. ForControls ::=

```
ForControl
| ForControls ';' ForControl
| ForControls ForCondition
```

def62. ForControl ::=

```
ForBinding
| ForCondition
```

def63. ForCondition ::=

```
'while' Expr 'else' Expr
| 'until' Expr 'then' Expr
```

def64. ForBinding ::=

```
ForName 'in' Expr
| ForName 'on' Expr
| ForName 'from' Expr
| ForName 'to' Expr
| ForName 'from' Expr ('to' | 'downto') Expr
```

def65. ForName ::=

```
Name [(':'  Type)]
```

def66. SwitchExpr ::=

```
'switch' Expr (SCase | SElse) 'endswitch'
```

def67. SCase ::=

```
('case' Expr)+ 'then' StatementSeq
```

def68. SElse ::=

```
'else' StatementSeq
```

def69. Type ::=

```
Expr
```

def70. TryCatchExpr ::=

```
'try' StatementSeq CatchSeq 'endtry'
```

def71. CatchSeq ::=

```
('catch' Arglist [('as')] StatementSeq)**
```

def72. UnitDef ::=

```
'define'
OpenModifier
('unit' | 'units')
UnitDef++
','
'enddefine'
```

def73. UnitDef ::=

```
Word [('=' Expr)]
```

def74. EnumDef ::=

```
'define'
OpenModifier
'enum'
Name
'='
Name++
','
'enddefine'
```

def75. Pathname ::=

```
[(Scheme)]
[(Root)]
[(Directories)]
[(Name)]
[(Suffix)]
[(Type)]
```

def76. Scheme ::=

```
Word ':'
```

def77. Root ::=

```
'//' Dotted
```

def78. Directories ::=

```
[('/')] (Dotted '/')++
```

def79. Name ::=

```
Word
```

def80. Suffix ::=

```
'.'  Word
```

def81. Type ::=

```
';' Word
```

def82. Dotted ::=

```
Word++ '.'
```