

# **SPL**

---

Design and evolution

Copyright © 2006 - 2007, Cybernetic Intelligence GmbH  
All Rights Reserved

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

### **Revision history**

1 January 2007	Initial draft.
15 February 2007	Corrected a few typos.

# Contents

<b>CONTENTS.....</b>	<b>3</b>
<b>PREFACE.....</b>	<b>5</b>
<b>THE TIMELINE.....</b>	<b>6</b>
<b>THE BIRTH OF SPL .....</b>	<b>7</b>
GUIDING PRINCIPLES .....	8
SPL LANGUAGE FAMILY .....	9
THE FIRST ATTEMPT .....	9
<i>Primitive types</i> .....	10
<i>Platform-independent types</i> .....	10
<i>Compound types</i> .....	11
<i>Pointers</i> .....	12
<i>Enumerated types</i> .....	13
<i>Derived types and type equivalence</i> .....	13
<i>Advanced entity attributes</i> .....	15
<i>Syntax</i> .....	16
<i>Heap control</i> .....	18
<i>Preprocessor</i> .....	19
<i>Standard library</i> .....	20
SPL COMPILER: INCREMENT 1 .....	21
<b>THE FIRST STEPS .....</b>	<b>23</b>
TYPE SYSTEM EXPERIENCE.....	23
IMPLEMENTATION PATTERNS .....	23
PROGRAMMING BY CONTRACT .....	25
SPL COMPILER: INCREMENT 2.....	26
<b>IMPROVING THE LANGUAGE.....</b>	<b>27</b>
THE STANDARD LIBRARY .....	27
SYNTACTIC SUGAR.....	27
TYPE QUALIFIERS .....	30
FEATURE DEPRECATION .....	31
INCLUDE CONTEXTS .....	31
REACHING THE LIMITS .....	33
<b>A NEW DAWN .....</b>	<b>34</b>
USE BEFORE DECLARATION.....	34
ENTITY REDECLARATION .....	35
BRINGING LABELS TO SCOPE .....	36
LABELS AS FIRST-CLASS VALUES .....	37
LEXICAL NAMESPACES .....	39
OPAQUE TYPES.....	41
ANONYMOUS FIELDS .....	43
EXTENDING THE PREPROCESSOR .....	44
REWRITING THE COMPILER .....	45
<b>REWRITING THE STANDARD LIBRARY .....</b>	<b>47</b>
EXTENSIBILITY .....	47
SCOPING THE LIBRARY .....	48

ENVIRONMENTS .....	50
GENERIC CONTAINERS .....	51
<b>COMING OF AGE .....</b>	<b>53</b>
<b>APPENDIX A: GNU FREE DOCUMENTATION LICENSE.....</b>	<b>54</b>
<b>APPENDIX B: SPL GRAMMAR (INCREMENT 1).....</b>	<b>61</b>
NOTATION.....	61
COMMENTS .....	61
LEXICAL ELEMENTS .....	61
LL(K) SYNTAX .....	63
<b>APPENDIX C: SPL GRAMMAR (INCREMENT 2).....</b>	<b>69</b>
NOTATION.....	69
COMMENTS .....	69
LEXICAL ELEMENTS .....	69
LL(K) SYNTAX .....	71

# Preface

This book is not a technical reference of the SPL language, library, or anything else that is SPL – related. It will not teach you how to program in SPL or any other language. What it will do instead is give historical perspective on how and why the SPL was born, how it developed over years, what decisions and why have been made to contribute to SPL design, and how the language is expected to mature further.

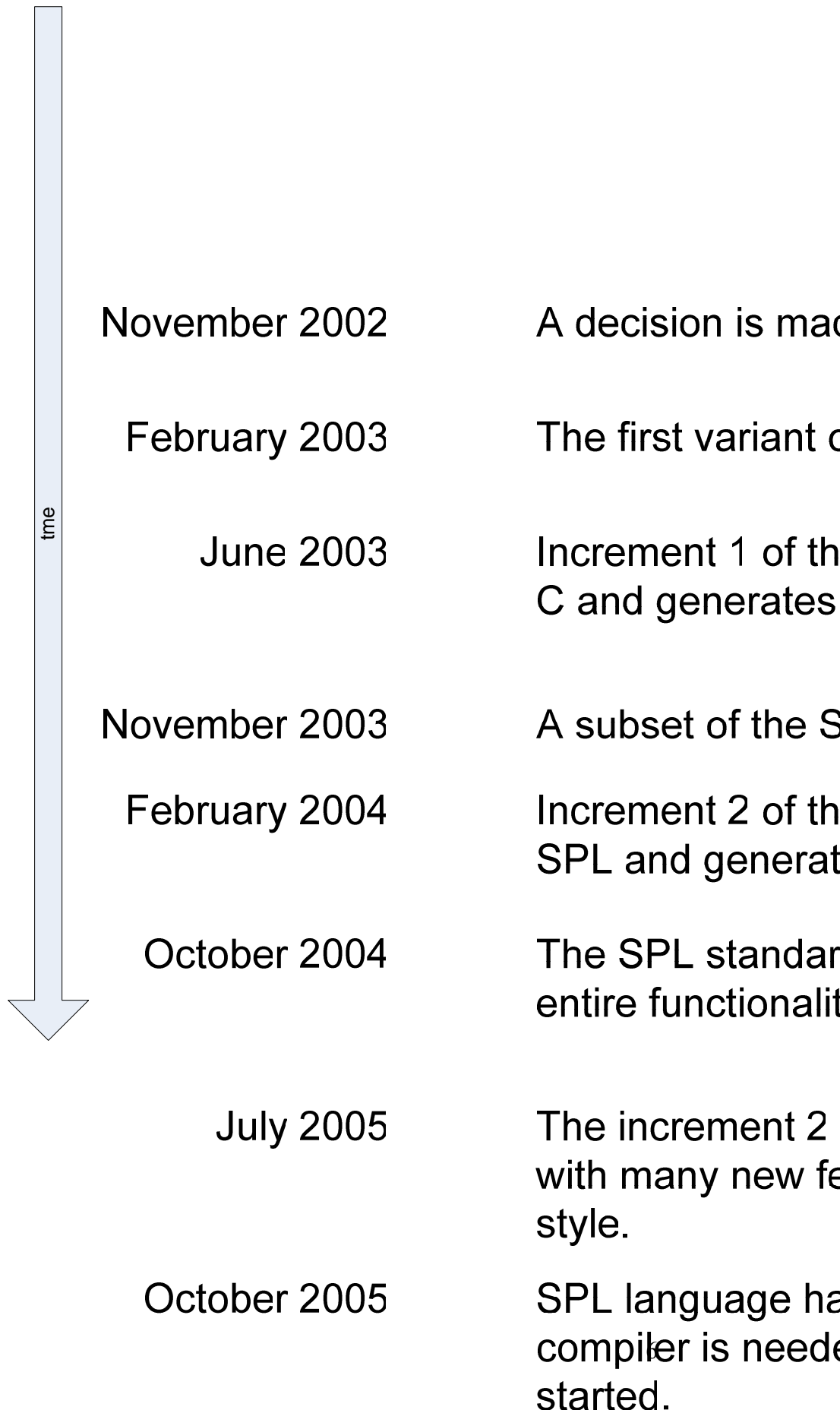
Why have I decided to write this book? The answer to that question can be credited to another book I have read some time ago. At the moment I was a highly competent software engineer, familiar with two dozen different programming languages – but I have accepted each one as a more-or-less static notation for expressing what I want my computer to do. Many of these languages have official (or unofficial, as may be) standards, revised very infrequently, so my approach at the time was to take whatever a language in question offered and to use it at best I can. The idea of a programming language as a living, breathing and changing entity was far from foremost in my mind – not least of all because I have never before taken part in designing a programming language myself (except for some rudimentary cases).

That is how things were when I first got hold of Bjarne Stroustrup’s book called “The Design and Evolution of C++”. Although I was quite proficient in both C and C++ at the time, reading that book was nothing short of a revelation for me. Knowing what features C++ offers to a programmer was one thing, but learning *why* these features appeared in the language, what with detailed discussion of the alternatives that didn’t quite make it gave me an insight I never had before. Moreover, these alternatives got me thinking outside C++ scope, which provided an invaluable help when I finally got to designing programming languages of my own.

For all these reasons, I have decided to write a similar book on SPL. Although a young language (at the moment of this writing the first “true” SPL compiler is still being finished), SPL has absorbed many important decisions made for equally important reasons. The language has had three prototype implementations over the years, each significantly different from the previous one, before stabilizing. I write this book in the hope that the evolution that led to what SPL is now will provide programmers with an invaluable insight as to *why* SPL has ended up as it did, as well as helping to improve SPL (and, maybe, other programming languages) further.

# The timeline

The following diagram illustrates the major events that took place in the SPL life.



# The birth of SPL

As far as I can remember, the SPL saga started in late 2002, when I have decided to develop an operating system of my own, having been equally fed up with both Windows and Linux – the then (and still) two most widely used operating systems in the desktop world. One of the choices I had to make quite early in the process of that development was to choose an implementation language.

The obvious first choice was to use either C or C++, not least of all due to stable language standards and widely available platform support. However, I was also fully aware of the problems that using either of these languages will bring in the long run.

One of the most important concerns was the lack of true portability of C/C++ code – having been involved in the development of C/C++ code that must build and run successfully on a multitude of severely different platforms for years I fully understood just how much effort is required to make such code truly portable.

My another concert was with C/C++ “weak” strong typing – although nowhere as permissive as that found in, say, PL/I (where the guiding principle can be expressed as “anything goes”), I have always regarded the C/C++ type system as a potential breeding ground for type errors, as opposed to the strict typing found in languages such as Ada or Eiffel.

Another major feature that I needed was programming-by-contract. Having previously participated in development of large projects in Eiffel, I was quite impressed by the Eiffel’s contract specification facilities, not least due to the fact that over two-thirds of all bugs in these projects were detected and reported by method pre- and post-conditions and fixed within minutes. I knew full well how to write C or C++ code that mimics programming-by-contract – and I detested the amount of effort required to do so.

Extending either C or C++ language to meet my needs was out of the question – both languages have international standards and, as far as I am concerned, the only reason to change a language standard is to get a better one – a process that I had no chance of initiating, let alone seeing through. Having briefly played with an idea of using some other language (like Eiffel) to implement the OS of my dreams, I decided to reject such an idea – any language I knew that was powerful enough for the job turned out to be too heavyweight for resource-critical code (such as OS kernel or device drivers).

All other options exhausted, I have decided to design a system programming language of my own and to use it for writing the new OS. For the time being I have decided to call the new language SPL (an abbreviation from “System Programming Language”); that name still stands today (although it has been suggested to me that it shall be made to mean “Structured Programming Language” instead).

Little did I realize that four years later my work on SPL will still be just coming to completion.

## Guiding principles

Starting to design a new system programming language, what I needed most was to keep the efficiency and low-level abilities of C. Lacking either would have eventually resulted in less-than-efficient OS. Moreover, lacking either will automatically mean that SPL will have no chance in being adopted by any other programmer but myself.

Having said that, I was more than willing to let go of the more cumbersome parts of the C syntax. Declarations in particular was the part of the syntax I intended to entirely rework – unlike C declarations, which must be read left-to-right, right-to-left, or even inside-out depending on the context, what I had in mind was something more Pascal-like.

Another component of the mix-to-be was language features that C lacks, programming-by-contract being one such feature. Other features were parameters passed by reference (you can do this in C but you have to do all the work yourself, working with pointers all over the code), platform-independent types (the range of all C basic types is platform-specific) and some syntactic sugar (like an ability to work with fields of the same structure or union variable without having to write the variable name all the time, an ability to initialize structures in one statement, etc.)

Having played around with several possible LL(k) grammars for the language being designed, I have come to formulate the first guiding principle behind SPL design:

**SPL shall be a language that provides a high-level syntax for well-defined low-level semantics.**

Having stabilized the first approximation of the SPL grammar, the work started on assigning semantics to various language constructs. Naturally, I had C semantics as a reference; however, I have very soon found myself disallowing many idioms common in C (such as default conversions) or extending C semantics where it was too restricted (such as array assignment). Moreover, the new SPL features that had no C equivalent have all turned out to be related to making the source code shorter and/or more verifiable. This goes directly opposite to C philosophy, where porting, verifying or debugging code requires extra effort from the programmer. If asked for a summary expression of these issues, I could formulate the second SPL design principle as follows:

**SPL shall be a language that requires programmer to expend extra effort if the programmer wants to write code that would be difficult to port, verify or debug.**

The final guiding principle was derived from the fact that it was a *system* programming language I was designing. The domain of system programming languages, once numerous, had long ago collapsed into just two languages – C and C++, and I wanted SPL to have a chance to be competitive there. Therefore:

**SPL shall be at least as efficient in both time and space as C.**

Note that the above rule does not mention C++. This is due to the fact that SPL was being designed to replace C, not C++, in the process of writing the OS of my dreams.

The final language design goal was something I did not have in mind at first – but it was something that became more and more important as the SPL implementation was developed:

**SPL shall allow construction of large, modular software systems from independently developed components.**

My experience has taught me over and over that, in the C world, construction of a large software system from pieces developed by independent sources more often than not leads to global name clashes and other visibility control issues. Needless to say, I wanted none of those to hinder SPL users.

All in all, the following quote from the SPL Language Reference describes my intent at the time:

*One could say SPL is what C should have been, were it not for the limitations of the time when C was designed.*

## **SPL language family**

As a long-time user of both C and C++, I have always been impressed with the ease of migration from the former to the latter – indeed, the majority of C code I have written over the years could be compiled with C++ with only minor modifications (or without any modifications at all). With that in mind, I have decided to develop SPL language family in three increments:

- First, the SPL language as such. This was (and still is) a direct counterpart of C as far as language features are concerned (except for those SPL features that are absent in C, such as programming-by-contracts – although C has rudimentary support for that in the form of the standard “assert” macro, SPL support for the programming-by-contracts is far richer).
- Next, the SPL2 language (or SPL+, or whatever else it ends up being called). This was to be a superset of the “proper” SPL in that it would provide all SPL facilities, as well as support for named namespaces (packages). This will make the language a direct feature counterpart of Modula-2 or Ada-95.
- Third, the SPL3 language (or SPL++, or whatever else it ends up being called). This was to be a superset of the SPL2 above, but will also support user-defined classes and interfaces, thus making the language a direct feature counterpart of C++.

At the moment of this writing, the work is still ongoing on the base SPL language. Whether SPL2 or SPL3 will ever materialize is uncertain; however, if they do, then both will be based on a better foundation than C was for C++.

## **The first attempt**

Having decided to pursue stricter semantics than that which could be found in C (e.g. that of Pascal/Ada), the natural choice of syntax for the SPL has originally been more Ada-like than C-like. With that in mind, there still were several syntactic differences introduced into SPL from the beginning and remaining there until this day. The

complete grammar of the “original” SPL can be found in Appendix B to this document.

## Primitive types

Being a dedicated proponent of strong type systems, it is no wonder that I have chosen a Pascal/Ada model of primitive types over that of C/C++. The standard C practice of using integers to represent character, boolean and enumeration values have, in my experience, led to a huge number of type errors; therefore SPL was to provide a dedicated type for each of them.

In its first edition, SPL was to provide the following primitive types: integer (both signed and unsigned), real, character and boolean, neither of which was to be type-compatible with each other. The latter rule was introduced to eliminate implicit type casts in order to make the programmer think twice before deciding on what type a particular variable shall have – a decision that had played out reasonably well even until the present day (analysis of the large SPL source code bases, specifically those forming SPL compiler and standard libraries, shows that type casts in a well-designed code are very infrequent and that every type cast records an explicit intent on the part of the programmer).

## Platform-independent types

One such syntactic and semantic difference was the introduction of platform-independent primitive data types.

My previous experience with numerous programming languages (C, Pascal, Ada, Modula-2, etc.) had more often than not hammered in the fact that primitive data types offered by a programming language are not as platform-independent as one might want them to be. Fortunately, I was also familiar with FORTRAN, specifically its sized primitive types, such as `INTEGER*1` being a binary integer exactly 1 byte long regardless of the platform. The decision to provide platform-independent primitive types for signed integers, unsigned integers and reals was an easy one to make, followed by realization that the same technique can also be applied to character types (such as `CHARACTER*4` representing 32-bit characters, which could be used for writing ISO-10646 – aware programs).

I have realized right from the start that always specifying the exact size of a primitive type would be too cumbersome for someone with a background in other programming languages, so the rule has been introduced declaring that specifying a type name (such as “integer”) without an explicit size should mean some platform-specific size. This would make programming techniques familiar to those who have previously programmed in Pascal, C, Ada, Modula-2, etc., where the exact range of a primitive type was implementation-specific. Originally, only `integer` and `cardinal` (another borrowing from Modula-2, denoting an unsigned integer) types were defined as platform-specific, with `real` always being the same as `real*8` and `character` always being the same as `character*1`; later on the two latter types will also be redefined as platform-specific.

Note that platform-independent type names (such as “integer\*1” or “real\*8”) are lexically a single keyword, i.e. spaces before and/or after an asterisk are not allowed.

An additional advantage of using explicit size specifiers instead of assigning each platform-independent type a different name (as, for example, does Java – the 1-, 2-, 4- and 8-byte integer types are called `byte`, `short`, `int` and `long` respectively) is in that when longer types or types with non-standard sizes are introduced into the language there is no need to invent new keywords that may clash with identifiers in existing code – one can easily extend SPL definition to include types such as `cardinal*16` (for 128-bit unsigned integers) or `real*10` (for 80-bit floating point values supported by Intel Pentium floating-point unit).

Someone familiar with C/C++ or MS .NET platform may wonder why platform-independent types specify the size in bytes (e.g. “integer\*1” for 1-byte signed integer) instead of bits (e.g. “integer\*8” for 8-bit signed integer). The answer to that lies in fact that SPL primitive types were specifically designed to map directly onto underlying hardware – and, although memory consisting of individually addressable 8-bit bytes is by far the most common in modern computer systems, there have always been exceptions to this rule, such as machines with 9-bit bytes or machines where whole words, not individual bytes, are addressable.

To accommodate all these situations, the size mentioned in an SPL type specification (such as “1” in “integer\*1”) refers to the number of consecutive addressable memory units occupied by the value. Therefore, on a machine with individually addressable 9-bit bytes an “integer\*4” value will be 36 bits long, whereas on a machine with memory consisting of  $2^{18}$  individually addressable 18-bit words an “integer\*1” value will occupy 1 addressable memory unit, i.e. 18 bits. Therefore, SPL programs written with platform-independent types are “truly” portable only between machines with the similar memory architecture. In practice, however, this does not present significant problem, as memories with individually addressable 8-bit bytes have all but pushed out other alternatives.

## Compound types

The set of compound types is more-or-less the same in all structured programming languages and includes arrays, records and variant records (unions in C). I saw no reason to change anything here, except I have decided to add several non-standard C features upfront rather than later. One of these features was an ability to declare a structure with the last field being a free array, as in (note that the original SPL syntax is being used):

```
type HeaderAndData is structure
    blockType : integer;
    dataSize  : cardinal;
    dataBytes : array [*] of cardinal*1;
end;
```

This kind of extension, which I was familiar with from my C experience, occasionally turned very useful when processing variable-length records. To make compound data

structures more orthogonal, I have explicitly allowed free arrays in unions as well as zero-sized data structures, so the following type declarations was (and its modern equivalent with newer syntax still is) a valid SPL:

```
type ArrayOfSomething is union
    i : array [*] of integer;
    c : array [*] of character;
    r : array [*] of real;
    b : array [*] of cardinal*1;
end;
```

The beauty of the data type declared above is in that it allows to treat the same area of memory as a sequence of integer, real, character or byte values – although not very type-safe, such facility is sometimes needed when writing a very low-level code.

## Pointers

From my C experience I knew that SPL had to have full support for pointers and pointer arithmetic.

What I did not want to do was to introduce something similar to C “void” type, which C programmer must use to declare typeless (raw) pointers “void\*” – it has always been my firm belief that if values of type “pointer to T” can exist, then so should values of type “T”. Other uses of “void” in C, such as specifying that a function has no arguments or returns no result, I have found equally superfluous – if a function has no arguments then specifying no arguments in the function header should be enough (this is the case in C++); similarly, if a function returns no value then not specifying the function return type shall be enough (this, unfortunately, could not be done in C or C++ because large amounts of legacy K&R C code rely on function return type being `int` by default).

With that in mind, I have adopted the C model of pointer arithmetic with three modifications. First, I have decided to use Pascal syntax “^T” for denoting the type “pointer to T” instead of C syntax “T\*” in order to allow type signatures to be read in strictly left-to-right manner. Second, I have introduced a new primitive type `pointer`, which could point to anything at all. Finally, I have decided to use a dedicated keyword constant `null` to represent a nil pointer rather than rely on preprocessor to define something similar to C/C++ `NULL` macro – I have seen too much code along the lines of:

```
char *s;
. . .
if (s != NULL && *s != NULL) . . .
```

to allow for this particular type error to occur in SPL.

To allow writing low-level memory manipulation code, where pointers are routinely converted to numeric values and back, the platform-dependent integral types `integer` and `cardinal` have both been defined to occupy exactly the same space as a `pointer`. To an SPL programmer this means that `integer<->pointer`,

`cardinal<->pointer` and `integer<->cardinal` conversions are all lossless.

Finally, the syntax for pointer dereferencing has also been lifted from Pascal. In SPL pointer dereferencing is a postfix operation ‘`^`’, not a prefix operation ‘`*`’ (as is the case in C). The C operation ‘`->`’, meaning “access a field of a composite pointed to” have been retained, mostly to make life easier to those with C background, although it does not introduce any new semantics in SPL as “`a->b`” is always equivalent to “`a^.b`” except for the fact that the latter looks unfamiliar to a C programmer.

## Enumerated types

The notion of enumerated types (as found in languages like Pascal or C) is so useful that I saw no reason not to have them in SPL. The two changes I have made was to disallow specifying explicit values for enumeration literals (in that respect SPL enumerated types are closer to Pascal than C) and specify the size of an enumerated value to be always equal to that of `cardinal*4` type, thus making enumeration ranges platform-independent.

It is interesting to note that at the time numeric values were assigned to enumeration literals starting with 1, not 0. The intent was to use 0 as a special “error” value, signalling that an enumerated variable has not been initialized. However, this was soon enough changed to C-style rules, where enumeration literals within an enumerated type are assigned their values starting with 0. The change has been made due to the frequent need to use enumerated types as array indexes, and array indexes in SPL start with 0.

## Derived types and type equivalence

Being familiar with type equivalence rules of many programming languages, I was fully aware that the current trend in programming languages is towards type equivalence by name (as opposed to structural type equivalence). I was also fully aware of the fact that any language using named type equivalence had constructs where type equivalence rules had to be relaxed in order to deal with (for example) constants of compound types. A typical example here is Pascal strings, where, for example, a string constant ‘`Hello`’ is assignment-compatible with any type declared as `array [1..5] of character`, regardless of how that type is named.

I don’t like special cases. What I needed is a type system that obeyed a simple and uniform set of rules. Given that some types used in the program are never named, I had no choice but to declare that structural equivalence is used for all types, including primitive, pointer, enumerated and compound types. Therefore, after the following declarations:

```
type T1 is structure
    x, y : integer;
end;
type T2 is structure
    x, y : integer;
end;
```

```
declare a as T1, b as T2;
```

the two variables a and b will be of the same type and, therefore, assignment compatible.

Now, that presented a problem – as a programmer I don't want my variables to be assignment-compatible with someone else's variables just because the types of the two coincide in structure. I was also fully aware of situations when values of primitive types are used to denote something else than the values comprising that type – typical examples being integer values used as file, window, process (and so on) handles. Another example is the situation where numeric values have implied measurement units, such as illustrated by the following C fragment:

```
int height = . . . ;
int weight = . . . ;
. . .
height += weight; // OOPS!
```

Fortunately, the simple solution has already been available in the form of Ada derived types, so the similar syntax was introduced into SPL:

```
type T1 is integer;
type T2 is new integer;
declare a as integer, b as T1, c as T2;
```

The SPL type derivation rules have been defined as follows:

- Any type N declared as “new T” has exactly the same range of values and allows exactly the same operations as T, except in the signature of these operations all occurrences of T are replaced with N. For example, if we declare type T as “type T is new integer”, then values of type T can be added with the binary “+” operator, yielding results of type T.
- For all derived types, name equivalence is used instead of structural equivalence.
- Values of derived types are never subject to default type conversions.

The introduction of derived types has allowed type errors such as those discussed above to be eliminated:

```
type FileHandle is new integer;
type WindowHandle is new integer;
declare file as FileHandle, window as WindowHandle;
file = window; // Type error detected by the compiler
```

```
type Height is new integer;
type Weight is new integer;
declare h as Height, w as Weight;
h = 0; // Type error detected by the compiler
h = Height(0); // OK, explicit cast
h += w; // Type error detected by the compiler
```

```

type Point is new structure
    x, y : integer;
end;
type Vector2D is new structure
    x, y : integer;
end;
declare p as Point, v as Vector2D;
p = w; // Type error detected by the compiler

```

As a result, SPL allows the programmer to explicitly use either structural or name equivalence for types on a per-type basis, with the syntactic difference between the two forms dressed in an Ada-like high-level syntax.

## Advanced entity attributes

My previous experience with several C++ toolchains have taught me the value of non-standard entity attributes (such as “`__declspec(dllimport)`”) used in MSVC++ to declare a procedure or variable dynamically imported from a DLL, or “`__attribute__((section("bar")))`” in GCC, which places a function or variable into a particular section). I wanted SPL to have these abilities. What I did not want is to populate attribute space with second-class citizens requiring some special syntax – after all, the section where a static variable is placed is just as important an attribute of that variable as its type or visibility. And, talking of visibilities, why should a public or private visibility be specified with a single keyword and export/import visibility require two?

For the above reasons, many features specified via preprocessor pragmas or other non-standard syntax in the C/C++ toolchains that I was familiar with were brought into SPL as proper entity attributes. These include:

- The “segment” attribute, that can be assigned to procedures and static variables in order to make sure they end up in the required segment.
- The “global name” attribute that can be assigned to procedures and static variables in order to make their global names (as visible by the linker) different from their names in the SPL source. This is particularly important when a program is developed in more than one language (for example, when most of it is SPL but some parts are written in assembler), as different languages may allow different characters within identifiers.
- The import/export visibilities for procedures and static variables. Although exporting a variable with a single keyword was by no means new (for example, in MSVC++ you could write “`__export int x;`” to declare a static variable that would be exported from the DLL where it is defined), the SPL innovation in declaring imported procedures and static variables is in that an import source (i.e. the DLL to import from) can be specified explicitly. For example, the declaration “`declare x as integer import "mylib.dll";`” will try to import `x` from “`mylib.dll`” at load-time. Later on, the explicit specification of an import source has been made optional to allow linker to resolve imports (that’s what existing C/C++ toolchains do with, for example, “`__declspec(dllimport)`”).

Another attribute that SPL allows to specify for global variables and procedures is an *alias*. From my previous experience with C++ implementations I was fully aware of name mangling and what it means to the compiler and linker. A typical name of a method or static field is normally presented to the linker in a form that is anything but readable – which is no problem until the linker must issue some diagnostics involving these names (such as a “double definition” error message).

The more-or-less standard way to cope with the situation (at least that’s how I have seen it done in several different C++ toolchains) is to teach linker to unmangle these names into something close to their original form in the C++ source. The problem I saw right there was in the fact that such linkers became too C++ – specific in that using them for linking object modules produced by some compiler that uses a different name mangling scheme will eventually result in meaningless diagnostics.

For that reason, I have first introduced the concept of aliases in a toolchain I have developed many years previously for Z80. In its basics, an alias is some string associated with a global symbol that is used instead of the symbol name when the linker must issue some message regarding the symbol in question; if a symbol has no explicitly specified alias then the alias is assumed to be the same as the symbol name. That decision had played out well at the time, especially as I played around with several different name mangling schemes to select the best one.

The main reason aliases made it into SPL was that right from the beginning I was expecting to use SPL as a target for compilers from other languages. When this is the case, it is essential to preserve original names of program elements as they appeared in the source program all the way through the SPL compiler to the linker – and assigning aliases to named entities of an SPL program does just that.

## Syntax

The original SPL syntax was much closer to Ada than C. Just to give an example, here is some SPL code written in the original SPL syntax:

```
type Byte is cardinal*1;
constant Pi is 3.1415;
declare x as public integer initial 0;

procedure gcd(a, b : cardinal) : cardinal
begin
    if a > b then
        a -= b;
    else if b > a then
        b -= a;
    else
        return a;
    end if;
end;
```

The remains of the Ada syntax occasionally show up in SPL to this very day, such as (for example) the use of the keyword “when” instead of “case” to denote switch alternatives (again, the original SPL syntax is used):

```

switch x is
  when 1: // Do something
    .
    .
    break;
  when 2 ... 4: // Do something else
    .
    .
    break;
end;

```

Another useful construct was a “with” statement, which allows working with multiple fields of the same structure or union without having to refer to an aggregate all the time. Pascal and Visual Basic both have similar facilities in that respect, but Visual Basic version has been chosen because for such an implicit qualification it uses a syntactic form different from that for a “normal” variable access. Hence, the SPL version became:

```

declare p as structure x, y : integer; end;
with p do
  begin
    .x = 1;
    .y = 2;
  end;

```

(Note that in Pascal the qualification operator ‘.’ (dot) would have been absent). Similar syntax was allowed for pointers-to-composites, mainly to make the language more orthogonal:

```

declare p as ^structure x, y : integer; end;
with p do
  begin
    ->x = 1;
    ->y = 2;
  end;

```

although it did not introduce any new semantics and could be rewritten as:

```

declare p as ^structure x, y : integer; end;
with p^ do
  begin
    .x = 1;
    .y = 2;
  end;

```

Next, I had to rework C logical operators. First of all, I needed the “implication” operator in both its full and short-circuited form (mainly for writing procedure postconditions at the time, e.g. “if *an attempt to read some bytes from a file succeeded* then *the actual number of bytes read does not exceed the limit*, etc.) Having played with an idea to use “=>” for full implication and “=>>” for short-circuited

implication I have quickly stumbled upon the fact that the use of the ‘^’ operator for pointer dereferencing would introduce ambiguities into the language, such as:

```
with p do a = x^.y; end;
```

The above assignment can mean “fetch the field *y* of the structure pointed to by *x* and assign that value to *a*” as well as “fetch the field *y* of the composite pointed to by *p*, bitwise-exclusive-or *x* with that value and assign the result to *a*”. Although the correct interpretation can be chosen based on exact types of *p* and *x*, I detested the idea of the same syntax denoting different semantics depending on the context where it was used. From there, it was not long before I have decided to use Ada approach where logical operators are written using keywords instead of special characters. An additional benefit was in that Ada logical operators already had both full (e.g. “and”) and short-circuited (e.g. “and then”) forms, the only missing thing was implication which promptly became “implies” and “implies then” operators.

Another difference between C and SPL is in the treatment of compound assignment operators, such as “+=” in “a += b”. Both C and C++ define a number of compound assignment operators (such as “+=”, “\*=” and “>>=”), each operator being a single token as far as the compiler is concerned (i.e. in C “a + = b” is a syntax error).

I wanted to keep the number of tokens to a minimum, so SPL allows any binary operator to be used in compound assignment in the form “a <binary operator> = b”. The key difference is in that the <binary operator> and an assignment operator “=” are lexically two different tokens, with any number of spaces and/or comments permitted between the two. The compiler is smart enough to correctly process most situations where these spaces are *not* there (e.g. if you write “a += b” SPL compiler will correctly treat “+=” as two consecutive tokens “+” and “=” because there is no token “+=” in the language); however, in more exotic cases the space must be written explicitly. For example, the expression “a = a < b” can be rewritten as “a < = b” but not as “a <= b”, since in the latter case the compiler will assume “<=” to be a single token and perform the compare operator instead of compare-and-assign.

Unlike C, SPL allows *any* binary operator to be used in compound assignment, so the statement “success and then = f(x)” is equivalent to “success = success and then f(x)”, which will evaluate *f(x)* and (possibly) update the value of the *success* variable only if it was already *true* before the statement. Although more exotic cases of the compound assignment (such as “compare-and-assign” or “short-circuit-logical-operator-and-assign”) are needed very rarely, they have been introduced into the language to make it more orthogonal, even at the cost of being slightly more difficult to implement than C.

## Heap control

Naturally, the very first alternative considered for SPL heap control was to use C-style `malloc` and `free` functions (or something with similar names). However, it immediately became apparent that, although easily usable in C, the approach will not

work in a strictly types language like SPL. The only reason `malloc` works in C is in that its return value is declared as “`void*`” and C permits implicit conversion of “`void*`” to “`T*`” for any type `T`. However, similar conversion from raw “`pointer`” to “`T^`” is not allowed in SPL, so the programmer will be forced to write an explicit cast each time memory was allocated from the heap. That just wouldn’t do.

Fortunately, an alternative solution was easily available – that offered by the C++ `new` and `delete` operators. Since SPL does not have the notion of user-defined constructors and does not permit operator overloading, SPL has ended up with a single global operator `new` that can be used to allocate both single objects and arrays:

```
declare p1, p2 as ^integer;
p1 = new integer;
p2 = new integer[100];
```

Similarly, as SPL does not have the notion of user-defined destructors, the `delete` operator does not need to know whether it deals with a single object or an array of object – all it needs to know is the address of the heap block to free, hence the array `delete[]` operator is not used.

It was fully understood even at the time that the advantage of having dedicated memory allocation functions (such as C `malloc` and `free`) is in that different memory allocation strategies could be defined by the programmer by writing his own functions with the interface similar to that of the `malloc/free` pair. For example, writing an OS kernel may well require the kernel private heap to be implemented for the kernel’s internal needs, with the functions `kernel_malloc` and `kernel_free` providing the necessary API. Needless to say that this is quite possible in SPL; however, an explicit type cast will be required each time `kernel_malloc` is called. Although not an ideal solution, this can be extended by using explicit memory allocation functions for each type:

```
procedure kernel_malloc_T() : ^T inline
begin
    return ^T(kernel_malloc(sizeof(T)));
end;
. . .
declare t : ^T = kernel_malloc_T();
```

The above solution works well in situations where only instances of a small number of types must be allocated from the custom heap. Note the “`inline`” property specifier applied to the procedure – my previous experience with C had made it clear that an ability to force procedure inlining was deemed so valuable that it became an integral part of the C++ standard.

## Preprocessor

To avoid reinventing a wheel, the C preprocessor was included into SPL in more-or-less its original form. The only new SPL preprocessor directive not found in C was the “`#once`” directive which, when used in a header file, prevents that header from being included more than once. The main reason for including this directive is a

constant need to guard C/C++ include files against multiple inclusion with conditional compilation directives, as in:

```
#ifndef file1_h_included
#define file1_h_included
. . . // Declarations comprising the header
#endif // ndef file1_h_included
```

Quite apart from having to write all these directives, there is always a remote chance that several independently developed headers will accidentally use the same identifier to guard against multiple inclusion. With the “#once” directive, life is much simpler:

```
#once
. . . // Declarations comprising the header
```

Note that the idea of the dedicated preprocessor directive to guard against multiple inclusion is by no means new – for example MSVC++ have long had the “#pragma once” preprocessor directive that has the same effect. The decision to make it a full-fledged directive in SPL instead of a pragma was based upon the need to provide as much portability as possible – hence SPL preprocessor does not, at present, have pragmas.

## Standard library

The original SPL standard library was in many ways similar to the C standard library, up to the point where the same function names were used for equivalent functions. However, what C standard library did *not* have was the language environment populated by a large number of platform-independent primitive types. Consider the following C example:

```
int x = atoi("12345");
```

This is all fine for C, where there exists only one integer type and only one character type. The need to allow similar conversion for long integer values caused another function to be added to the C standard library:

```
long int x = atol("12345");
```

That works reasonably well for C – but SPL has 10 integer types and 4 character types, which would result in 40 string→integer conversion functions, all with different names. Trying to remember all these names, let alone the fact that all these functions perform similar conversions and differ only in parameter and/or return types, would be an impossible task. C++ - like templates were an obvious solution, but SPL did not have templates at the time and I did not want to make the language more complicated by introducing them. The same can be said about procedure overloading – which will be insufficient anyway as some `atoi` variants will have the same parameter types and differ in return type only.

Failing that, the next obvious choice was to use the same base name for all functions with the similar purpose but to suffix this base name with the description of each

concrete function parameter and/or return type. Hence the string->integer conversion function family will consist of functions like:

```
procedure atoi(s : ^character) : integer;
procedure atoiS1(s : ^character*1) : integer;
procedure atoiS2(s : ^character*2) : integer;
.
.
.
procedure atoiI1(s : ^character) : integer*1;
procedure atoiS1I1(s : ^character*1) : integer*1;
procedure atoiS2I8(s : ^character*2) : integer*8;
```

Note that the suffix denoting the type of the return value is the last suffix used, because the return type of a procedure is specified *after* the parameter list in the procedure header.

Note also that a programmer who does not use platform-independent primitive types does not need to use suffixed versions of the SPL standard library services at all – hence in a program that uses `character` variables to store characters and `integer` variables to store integers the conversion will be performed as:

```
declare x as integer initial atoi("123456");
```

with the full understanding that on 8- and 16-bit platforms the conversion result will be truncated for being out of `integer` range.

## SPL compiler: Increment 1

The very first SPL compiler was written in the first half of 2003. It was written in ISO C 89 and could parse SPL programs and generate ISO C 89 output. The result would then be processed by a C compiler to obtain an object file that can be linked into an executable module.

The choice of C as an output language was more or less governed by the same reasons as those which caused the initial C++ implementation (cfront) to write C instead of machine code – C was the most portable high-level assembler around. All code generation and optimization decisions could then be delegated to the C compiler, whereas the SPL compiler will be mostly concerned with SPL syntax and semantics.

The fact that the generated code would not be of the highest quality (because some SPL features have no direct C counterpart and had to be emulated with library support, the most important of these features being nested procedures and accessing nonlocal nonstatic data from enclosing scopes) was, at the time, irrelevant – the purpose of the SPL compiler was to be useful in:

1. Developing enough of the SPL standard library to allow rewriting SPL compiler itself in SPL.
2. Locating any deficiencies and oddities in the SPL language.

For the same reason, the compiler implementation language was C and not C++, as it was apparent that migrating C code to SPL would be a straightforward exercise, while

migrating C++ code to SPL would present a much larger challenge of reducing compiler abstraction level during migration.

The first increment of the SPL compiler became operational in mid-2003 and consisted of ~45000 lines of C code. Since then it has been superseded by increments 2, 3 and, finally 4 (the latter represents the first compiler release consistent with the “true” SPL language as defined in the SPL language and standard library specifications); the sources of the 1<sup>st</sup> increment have since been discarded. Roughly, the 4 SPL compiler increments can be characterized as follows:

- Increment 1 – the compiler written in C and accepting the “old” SPL as its source language.
- Increment 2 – the compiler written in “old” SPL and accepting the “old” SPL as its source language.
- Increment 3 – the compiler written in “old” SPL and accepting the “true” SPL as its source language.
- Increment 4 – the compiler written in “true” SPL and accepting the “true” SPL as its source language.

Further increments of the SPL compiler will be discussed in more detail in the corresponding chapters of this book later on.

## The first steps

Once the first SPL compiler has been more or less finished, the logical next step was to develop enough of a library support to allow the SPL compiler itself to be rewritten in SPL.

Once I have started writing SPL library, several things became quite apparent.

## Type system experience

The first one, to my surprise, was the amount of type-unsafe code I was writing. Being a dedicated proponent of strongly typed languages, my practical experience with these languages was nevertheless limited to some Pascal and Ada work I have done more than 10 years ago and some Eiffel work later on. As the languages I have used most over many years were C and C++, my initial attempts to write SPL code were full of type errors caused by programming idioms that would have been safe in C/C++. Just a few examples include routine assignment of characters to integers (and vice versa), comparing signed values with unsigned values for both equality and order, writing mixed-type expressions... the list can be continued. Needless to say, relaxing SPL type system was out of the question – after all, I have made it as strict as possible on purpose – so the only choice was to take enough care to avoid writing type-erroneous code.

Ironically, my SPL experience had a backlash in that respect – my C/C++ coding habits have since gradually changed to match SPL strict typing rules, so that nowadays you will not find mixed-type expressions in my C/C++ code (for example, given an integer variable `x` it is a typical C/C++ trick to write code along the lines “`if (x) ...`”, whereas the type-safe form would be “`if (x != 0) ...`”, which is just as efficient).

## Implementation patterns

Having previously implemented a standard C library for a 8086 toolchain, I eagerly started writing the SPL standard library along the same lines... only to discover that I had some ways of thinking to leave behind.

A typical illustration would be an `atou` service that converts a character string into an unsigned integer value and returns the result. The obvious implementation would run along the following lines:

```
procedure atou(s : ^character) : cardinal
begin
    declare value as cardinal initial 0u;
    // Skip initial spaces
    while isspace(s^) do s++;
    while isdigit(s^) do
    begin
        value = 10u * value + cardinal((s++)^ - '0');
    end;
    return value;
```

```

end;

procedure atouS1(s : ^character*1) : cardinal
begin
    declare value as cardinal initial 0u;
    // Skip initial spaces
    while isspaceS1(s^) do s++;
    while isdigitS1(s^) do
        begin
            value = 10u * value + cardinal((s++)^ - a'0');
        end;
    return value;
end;
// ... and so on

```

So far so good – except for the fact that SPL has 4 character types and 5 cardinal types. That makes a total of 20 conversion procedures, each one different from its brethren by but a few tokens. I did not want to write all this duplicate code.

One obvious way out was to simulate parameterized procedures with preprocessor facilities. I will not describe just how ugly the solution would have been – it suffices to say that I wanted to use as little preprocessor as possible.

Instead, the approach I have used was to write generic services parameterized with callbacks. Returning to the previous example, the `atou` family of functions would be implemented as:

```

type GetCharProc : procedure : character*4;
type UngetCharProc : procedure(* : character*4);

procedure _atou(
    getchar : GetCharProc,
    ungetchar : UngetCharProc
) : cardinal*8
begin
    declare value as cardinal*8 initial 0ul,
           c : character*4;
    // Skip initial spaces
    c = getchar();
    while isspace(c) do c = getchar();
    while isdigit(c) do
        begin
            value = 10ul * value + cardinal*8(c - i'0');
            c = getchar();
        end;
    ungetchar(c);
    return value;
end;

procedure atou(s : ^character) : cardinal
begin

```

```

    procedure getchar : character*4
    begin
        return character*4((s++)^);
    end;
    procedure ungetchar(* : character*4)
    begin
        s--;
    end;
    return cardinal(_atou(getchar, ungetchar));
end;

procedure atouS1(s : ^character*1) : cardinal
begin
    procedure getchar : character*4
    begin
        return character*4((s++)^);
    end;
    procedure ungetchar(* : character*4)
    begin
        s--;
    end;
    return cardinal(_atou(getchar, ungetchar));
end;
// ... and so on

```

Although it may seem like an overkill for this particular case, the actual SPL library services tend to be somewhat larger (for example, the actual `atou` family of SPL library services had to deal with different radices as well as report parsing errors, whereas the `printf/scanf` family of SPL library services includes a huge amount of formatting code that is common for all type-specific variants. To give an example, increment 2 of the SPL standard library contained 31 `printf`-style functions, the longest of which was 54 lines long, whereas the parameterized procedure used to implement all of them was 2732 lines long).

To be sure, I have had my share of negative experience with callbacks in C – not the least nuisance being the need for the callback to know what data it was supposed to be using. The typical C solution would be to pass an extra parameter to the callback-aware service, so that this extra parameter can be used by the callback when the need arises. However, SPL eliminated the need for the said extra parameter by supporting nested procedures – note that callbacks in the above example use and modify the nonlocal variable `s`, which is a parameter of an enclosing procedure.

## Programming by contract

Another issue was SPL support for programming-by-contract. From the very beginning I have saturated SPL library services with preconditions, postconditions and assertions – and as the SPL library grew the number of bugs detected by these constraints grew proportionally. This I was familiar with from C. What I was not quite expecting was the fact that all but a few constraint violations were reported as precondition or postcondition failures rather than assertion failures. The fact that a C

programmer can only use assertions (in C declarations must precede statements in a block) suggests that C may have a serious vulnerability in that respect.

There may be a logical explanation for that which, at least as far as I can see, runs along the following lines: a typical library service is a relatively short procedure, easily grasped by the programmer such as myself as a whole. Within such a small procedure it is fairly easy to remember not to do anything stupid, and assertions are just another way to make sure you don't. On the other hand, when a library procedure calls several other library procedures (and these can in turn call something else as well), it's far easier to forget that some other procedure somewhere deep in this call chain has a precondition.

Incidentally, when it came to rewriting SPL compiler itself in SPL, the history repeated itself – there were far more bugs detected by precondition and postcondition failures than by assertions. At the extreme end, in the increment 3 of the SPL compiler all constraint violations were precondition and postcondition violations – no assertion faults have ever been triggered.

## **SPL compiler: increment 2**

The work on an increment 2 of the SPL compiler started in late 2003 and consisted mostly in a statement-by-statement translation of the increment 1 compiler sources from C to SPL. As SPL is semantically a superset of C (at least as far as the language is concerned), the migration occurred with very little difficulty – the only problem I remember was in that the part of the SPL standard library I had implemented at the time did not have a counterpart to C `set jmp/long jmp` functions, so the error recovery functionality had to be modified in a few places. By February 2004 the migration has been complete and increment 2 of the SPL compiler has converged (i.e. taking a compiler's executable and using it to compile the compiler's own source and standard library produced exactly the same executable).

Just like increment 1, the increment 2 of the SPL compiler consisted of about ~45000 lines of SPL code and could only generate C as output. Eventually, it will grow to ~68000 lines of SPL code before being superseded by increment 3.

## Improving the language

Once an increment 2 of the SPL compiler was able to compile itself, a process of polishing the language has started. Over the next year-and-a-half the SPL language will acquire many of its present features, with the SPL standard library further developed in parallel.

### The standard library

At the time when the increment 2 of the SPL compiler was ready, the SPL standard library was far from finished – in fact it more or less included only those features needed by the SPL compiler.

Now it was time to finish the work. Taking the C standard library as an example, I have tried to enrich the SPL standard library to the point that it provides everything the C standard library does. This meant implementing a lot of extra functionality (such as date/time services, searching and sorting, console I/O, directory and file control services, etc.) Some of the features of the SPL standard library were created without a corresponding CRT counterpart – notable examples include support for character sets, ISO 639/ISO 3166 API and process-local storage. The general rule was to include features that are needed to properly implement the SPL standard library (for example, ISO 639/ISO 3166 API was needed to implement proper locale support, character sets were needed to properly handle textual I/O, etc.)

Although greatly extended over the next year-and-a-half, the 1<sup>st</sup> increment of the SPL standard library had never reached its intended functionality. Among features that were never implemented were mathematical routines (such as `sin()`, `sqrt()`, etc.), thread API and nonlocal jump routines (similar to CRT `set jmp/long jmp` functions). The main reason for missing all that functionality is in that by the time I was ready to implement it, I have already made a decision to write increment 3 of the SPL compiler, that would introduce many additional features into the SPL language – once again it was more important to write an SPL compiler than to work on the SPL standard library (in addition, new features introduced into the SPL language would make some of the not-yet-implemented standard library routines obsolete, a typical example being nonlocal jumps to replace `set jmp/long jmp` routines).

### Syntactic sugar

It is an interesting observation that, although semantics of SPL remained quite stable until it was time for the increment 3 of the SPL compiler, the syntax of the language has undergone some significant changes during the same period. The main effort was directed towards ensuring syntactic consistency of the language.

One of the first syntactic changes was the replacement of `begin ... end` block delimiters with `{ ... }` curly braces. During one of my earlier projects (not related to SPL) I had to design a language that allowed both `begin ... end` keywords and `{ ... }` braces as block delimiters (there were two stakeholders, each wanting a particular style, and none willing to give in), so for a short time SPL allowed both `begin ... end` – style blocks and `{ ... }` – style blocks; however, soon enough the `begin` and `end` keywords were removed from the language, leaving SPL blocks with a C syntax.

Another change was in the use of separators in entity declaration. Original SPL syntax mandated Ada-style keyword separators, as in:

```
type Weight is new integer;
constant Pi is 3.1416;
declare x as integer initial 0;
```

The problem with the latter form (variable declaration) is in that two different syntactic forms were used to assign types to variables:

```
declare point as structure { x, y : integer; };
```

Note the use of the `as` keyword to separate variable name from variable type at an outer level – as opposed to using a colon to separate field name from a field type.

One alternative was, naturally, to use the keyword `as` consistently for all name/type bindings, as in:

```
declare point as structure { x, y as integer; };
procedure sqrt(x as real) as real ...
```

In my subjective opinion, that was just way too similar to MS Visual Basic. The only other alternative was to go the other way around:

```
declare point : structure { x, y : integer; };
procedure sqrt(x : real) : real ...
```

Having reworked the variable declaration syntax, I now had a situation where some declarations used keywords as name/properties separators, while other declarations used punctuation:

```
type Weight is new integer;
constant Pi is 3.1416;
declare x : integer initial 0;
```

Having briefly played with an idea of using the same character (colon ‘:’) as name/properties separator for all declarations, as in:

```
type Weight : new integer;
constant Pi : 3.1416;
declare x : integer initial 0;
```

I have instead decided to use a colon ‘:’ for declarations where names are assigned properties, and an assignment operator ‘=’ for declarations where names are assigned values:

```
type Weight = new integer;
constant Pi = 3.1416;
declare x : integer initial 0;
```

A short time after that, I realized that a variable declaration can specify *both* properties of that variable *and* its initial value, so the `initial` keyword was a goner, with variable initialization using C syntax instead:

```
declare x : integer = 0;
```

Another syntactic simplification was removal of keyword abbreviations from the language. Originally, SPL followed the PL/I model of allowing some of the longer keywords to be abbreviated, as in:

```
dcl point : struct { x, y : integer; };
const Pi = 3.1416;
proc sqrt(x : real) : real ...
```

However, after rewriting SPL compiler itself in SPL, I have noticed that I did not use these abbreviations in either compiler or standard library sources. To make the language that much smaller, abbreviated keywords were removed from the language.

Yet another syntactic simplification was in free array type signatures. In several contexts, SPL allows the use of an asterisk ‘\*’ to denote the fact that something is “not there”, as in missing parameter or field names:

```
procedure boo(* : integer) ... // parameter has no name
type T = structure
{
    a : integer;
    * : character;           // field has no name
    c : real;
};
```

Originally, when declaring a free array an asterisk had to be used instead of an array dimension, as in:

```
declare x : external array [*] of integer;
```

To make the code more familiar to C programmers, this has been changed to:

```
declare x : external array [] of integer;
```

At the same time, the syntax for multidimensional arrays was brought closer to its C counterpart. While original SPL required the full type signature for each array dimension, as in:

```
declare x : array [10] of array [20] of real;
```

the syntactic shortcut was introduced that allowed the C-like shortened form:

```
declare x : array [10][20] of real;
```

All in all, as a result of syntactic changes SPL was starting to look much more like C (and, to some degree, Pascal) than Ada. The complete grammar of the “improved” SPL can be found in Appendix C to this document.

## Type qualifiers

At some moment during SPL language enhancement, it dawned on me that, which C allowed qualifying types with `const` and `volatile` modifiers, SPL did not – the most apparent reason being in that Ada did not have type qualifiers either, and SPL type system was modelled after that of Ada. I decided to correct that oversight, the only difference between C and SPL type qualifiers being the use of `constant` keyword instead of `const` to declare unchangeable types (this presented no problem, as `constant` was already a reserved word in SPL). While the meaning of the `constant` qualifier was straightforward enough, the SPL `volatile` is quite different from a C `volatile` – while the latter means “a value that can be changed by something else than the program”, the former means “a value that can be changed by something else than the currently executing thread”, thus mandating SPL `volatile` variables as safe for inter-thread communication (this would, probably, have been the case for C as well, were it not for the fact that C had been created in a single-threaded world).

An idea to introduce one more qualifier came up after I have spent some significant amount of time writing C++ code that had to use the same structure layout on both 32-bit and 64-bit platforms. The thought at the time was “wouldn’t it be nice if a structure was laid out exactly as I specify, without compiler introducing some weird padding between the fields” – in the C++ project I mentioned above this was eventually achieved via packing pragmas (different for each toolchain we had to support); fortunately, I remembered that Pascal allowed declaring structures and arrays as `packed` “in order to save space”. As SPL arrays were already laid out similarly to C arrays (i.e. no padding between consecutive array elements), I have originally allowed only structures and unions to be declared as `packed` (meaning that fields were laid out one after another without any padding), only to find out that multi-byte fields were now no longer naturally aligned. While not a problem in IA-32 architecture (which is where SPL development took place), I did not want problems to surface when porting SPL to platforms that require natural alignment for multi-byte items, so the semantics of the `packed` type modifier was changed to mean that:

- The `packed` modifier can be applied to any type,
- Variables of type `packed T` have the same range as variables of type `T`,
- Variables of type `packed T` occupy as little memory as possible, and
- Variables of type `packed T` do not necessarily obey alignment constraints of type `T`.

To make things consistent, I had to mandate that if `T` is a composite type, that all components of a type `packed T` are themselves `packed` (e.g. all fields of a `packed` structure are themselves `packed`, as any of these fields can violate natural alignment constraints).

The addition of `packed` types to the language allowed individual types to be laid out in either time-efficient or space-efficient manner. I played briefly with the idea of making types `T` and `packed T` assignment-compatible, ending up with allowing assignment-compatibility between packed and unpacked primitive types only (but not structures, unions or arrays). This is, to this day, one of the few remaining cases when SPL treats primitive types differently from composite types – the main rationale being that making `T` assignment-compatible with `packed T` may lead to type errors, as `^T` is not assignment-compatible with `^packed T` (because `T` and `packed T` generally have different sizes).

## Feature deprecation

From my previous experience with Java and Eiffel, I knew that there was a case for being able to declare entities as “out-of-date”. Even in languages where the concept of “out-of-date” features is not explicitly present, there is some drive towards migrating from older features to newer ones (for example, the GNU C++ toolchain will issue a warning if a `tmpnam()` function is used, saying something along the lines of “`tmpnam()` function is dangerous, use some better alternative”).

I wanted SPL to be useful for (among other things) writing reusable libraries – and these are known to change with time, part of the change being adding new things and dropping old ones.

The solution was simple enough – as declarations of SPL named entities already specified properties of these named entities, adding a `deprecated` property was not a difficult change to make. It also was simple enough to teach the compiler to issue a warning whenever a deprecated entity was being used, with only a single nuance – a deprecated entity is only “truly” deprecated when used by something which is *not* (otherwise too many “deprecated entity in use” warning were generated).

From reading numerous books on a software life cycle, I was fully aware that objects do not always follow the “current, then deprecated, then gone” life cycle. Many sources agree that the somewhat more lenient “current, then deprecated, then obsolete, then gone” life cycle shall be used instead, where “deprecated” means “will be gone within the next 10 years”, and “obsolete” means “will be gone within the next 5 years”, or something similar. I have thought about allowing SPL entities to be declared as either `deprecated` or `obsolete`, and have eventually decided against that. The main argument here was “if an entity goes through more than one state on its way to oblivion, why limit it to just two?” Indeed, one can easily imagine a three-state removal process (e.g. “current, then deprecated, then obsolete, then hidden, then gone”), as well as something far longer. I did not want to complicate the language, so in SPL `deprecated` means “don’t use this in newly written code”.

## Include contexts

As mentioned before, the original SPL preprocessor was more-or-less direct counterpart of the C preprocessor (apart from an absence of pragmas in SPL preprocessor – a decision that was made for the sake of portability and stands to this day).

There was, however, one aspect of the C preprocessor that I was not happy with – the `#include` directives. The C preprocessor uses a special syntax for file name strings appearing there, as well as allowing either quotes `" "` or angle brackets `<>` to be used as string delimiters. Originally, SPL mandated that include file names shall be SPL string constants; however the include file lookup rules had to be complicated compared to those of C, as the same form of an `#include` directive was now used for both system and user headers.

The solution to the mess was not apparent until some time later, when I was involved in an independent component-based C++ project. Individual components were developed by different teams, ending up with several C++ headers with the same name existing in different directories. After spending several days trying to get include path lists in proper order, I remember thinking “wouldn’t it be nice if I could specify *where* to include from”. Of course, one could always write fully qualified header file names – only this would have failed miserably in a cooperative development environment where the working directory is different for each developer. Instead, the concept of include context has been invented.

In SPL, an include context is a named list of directory names. Normally, when an `#include` directive is written:

```
#include "io.sph"
```

it is up to the SPL preprocessor to decide whether to look up for the header file in system or user places. The lookup rules are further complicated by the fact that the list of directories where the header is searched for can be different depending on where the `#include` directive occurs.

With what eventually have been called qualified includes, the name of the include context is specified explicitly:

```
#include "io.sph" from SPL_STANDARD_LIBRARY_HEADERS
```

In this case, the header file `io.sph` is searched for *only* in directories associated with the include context `SPL_STANDARD_LIBRARY_HEADERS` and nowhere else. This greatly simplifies an assembly of programs from independently developed components – in order to avoid inclusion of wrong headers all that is necessary for the developers is to assign a unique include context name to each component; at that, each developer can have different working directory structure without the need to modify the SPL sources.

Once qualified includes were introduced into the language, the include context name `SPL_STANDARD_LIBRARY_HEADERS` was immediately reserved to mean “only SPL standard library headers” – an `#include` directive such as the one above will always include a standard library header even if the user has written his own private header `io.sph`.

## Reaching the limits

As the SPL compiler was being developed, the list of “would be nice to have” features grew. While some of these features remain un-implemented to this day (such as bit fields), others were important as they would be useful in the compiler itself.

Continuing to further extend the increment 2 of the SPL compiler was out of the question – some of the features I needed were impossible to handle with the existing compiler architecture (such as allowing entities to be declared after the first use, which is not achievable in a one-pass compiler). Given that, I had no other choice than to rewrite an SPL compiler from scratch once again. The increment 3 of the SPL compiler was first made to work in early 2006, eventually growing to ~116000 lines of SPL code before being superseded by increment 4.

## A new dawn

Before the work on the next increment of the SPL compiler could begin, I had to decide what new features will make it into the SPL language, and in what form. Significant time has been spent in going over the “would be nice to have” features list, which was by then quite sizeable.

### Use before declaration

One of these “nice to have” features was an ability to declare procedures after their use as opposed to having to declare them before use. Increment 2 of the SPL compiler contained a large number of recursive syntactic and semantic routines, and the amount of forward procedure declarations was getting on my nerves. At about the same time, my work on a separate C++ project which contained several dozen of inter-dependent headers taught me just how much time can be spent on arranging `#includes` in an order that would compile, let alone that sometimes this is impossible. Just as an example, imagine the two C++ classes like these:

```
class A { ... B foo() { ... } };  
class B { ... A bar() { ... } };
```

No matter in which order the two class declarations appear in the compilation unit, it is impossible to declare both `foo()` and `bar()` as inline functions. While exactly the same situation cannot exist in SPL (as it does not have classes), similar problems could occur with mutually recursive inline procedures, or inter-dependent data types declared in different headers.

One more argument to allow entities to be declared after the use was in the fact that the language already has a whole class of entities that were routinely declared after use – namely statement labels. I wanted an orthogonal language, but the idea of forcing the programmer to declare labels before they can be used (as was the case in early Pascal) was one I totally detested. The only other way to achieve an orthogonal behaviour was to allow *all* entities to be used before declaration (which would also allow interdependent headers to be `#included` in any order, but I did not realize that at the time).

I was familiar with languages that allowed things to be used first and declared later (such as PL/I), and I was fully aware that such languages were things of the past, with current trend being towards the other extreme, where everything must be declared before use. I was also aware that such “declare everything before use” languages actually relaxed the declare-before-use rule in certain context (for example, in C++ an inline method of a class can use a data member of the same class even if the said data member is declared after the inline method – the “declare everything before use” rule does not work within a single class).

As things are, it turned out that allowing entities to be declared after use did not introduce a major programming hazard – after all, the mere fact that a programmer *can* use a variable a thousand lines before it is declared does not mean that the programmer actually *will* write such code. My own experience with declare-after-use, gained while re-writing both SPL compiler (that would be increment 4) and SPL

standard library was in that it eliminated a lot of forward declarations, and that the only difficulty it introduced into the language was the difficulty of implementing this feature in the compiler (but, then, it is my firm belief that programming languages shall be made convenient for language users, not for compiler writers).

## Entity redeclaration

From my experience with C and C++ I was quite familiar with “multiple declarations, one definition” rule, which requires that every named object can be declared (i.e. its properties specified) several times, but defined (i.e. actually instantiated) only once. So far so good – except in C both constant and type declarations are actually definitions, which causes a lot of the guarding by preprocessor, such as:

```
#ifndef _SIZE_T_DEFINED
    typedef unsigned int size_t;
    #define _SIZE_T_DEFINED
#endif
```

Quite apart from the fact that I don’t like to use preprocessor facilities, I fail to see the point in disallowing type redeclarations, such as:

```
typedef unsigned int size_t;
...
typedef unsigned int size_t;
```

as long as they do not introduce any ambiguity.

The “original” SPL allowed non-ambiguous redeclaration of types, constants and procedure prototypes (but not variables or procedure definitions), so the following SPL code would compile properly:

```
constant Size = 10;
type String = array [Size] of character;
procedure initialize(s : out String);
...
constant Size = 10;
type String = array [Size] of character;
procedure initialize(s : out String) ...
```

I wanted the same treatment for all named objects. After some serious thinking, I have redefined semantics of entity declarations while retaining their syntax. In the new semantics, a declaration of an entity (whether constant, type, variable, procedure or label) specifies some (not necessarily all) properties of this entity. Several declarations in the same scope can specify properties of the same entity, provided there is no conflict. This interpretation of entity declaration meant that all the SPL legacy code (such as the one above) would still compile; however, it also allowed declarations to be split into several chunks, as in:

```
declare x : integer;
declare x : public;
declare x : deprecated;
```

```
declare x : = 10;
```

which would be semantically equal to a single declaration:

```
declare x : deprecated public integer = 10;
```

Although this example is a little bit weird, an ability to declare the same global variable in more than one header eventually came in handy.

## Bringing labels to scope

Looking to further improve SPL orthogonality, I could not help but stumble upon the fact that SPL statement labels obeyed scoping rules different from those used by other named entities (such as types, constants and variables). This was largely a C legacy, where it is permitted to `goto` from outside a block to a label that is within the said block. Quite apart from introducing a special case into the language scoping rules, this gave way to a multitude of unsafe programming constructs (such as `goto`-ing into the middle of a loop body). I don't like special cases, and I don't like unsafe.

A short experiment was carried out to see if changing label scoping rules to match those used by everything else was undertaken, with surprisingly good result – not a single existing SPL source was broken by the new label scoping rules (the SPL code base was, naturally, SPL compiler itself and SPL standard library). Hence, the language now had uniform scoping rules, although that meant that several labels with the same name could now exist in different blocks, as in:

```
{
  Boo: x = 1;
  goto Boo; // Goes to the "Boo" above
  {
    constant Boo = 2;
    goto Boo; // ERROR – in this scope "Boo" is not a label
    {
      Boo: y = 3;
      goto Boo; // Goes to the "Boo" above
    }
  }
}
```

Although the above code looks outright weird, there is no ambiguity introduced, as static scoping rules can always resolve an identifier to a particular label.

The only thing I still had to tackle was nonlocal jumps. The reasoning went along the following lines: if an inner procedure can use type, constant or variable declared in an enclosing outer procedure, why should the language disallow jumps from an inner procedure to a label declared in an enclosing outer procedure? I knew several languages that had that behaviour (PL/I and Pascal being just the two examples), I knew that implementing nonlocal jumps would be costly (but no more costly than using `set jmp/long jmp` standard library routines in C), and I also knew that this would make a convenient alternative to exception handling, which SPL does not have

(to be precise, the use of nonlocal jumps to emulate exception handling did not come into a full bloom until `label` variables were introduced into the language – with `label` variables it is possible to perform nonlocal jumps between unrelated procedures, whereas without `label` variables nonlocal jumps are allowed only from inner to enclosing outer procedures).

## Labels as first-class values

Looking further at labels, I realized that they were now the only entities in the language for which constants could be specified but variables could not. For example, SPL allows both integer constants and integer variables to be declared; similarly, procedures can be viewed as constants of procedure types, with variables of procedure types implementing the C-like pointer-to-function behaviour. However, there were no `label` variables – only `label` constants, syntactically specified as statement labels.

Fortunately, I was quite familiar with PL/I, where `label` variables are supported. An introduction of a new primitive type `label` into the language was an exercise of but a few minutes; however, it took far longer to hammer out all the consequences of that change.

One of the consequences was caused by the fact that SPL has `break` and `continue` statements, which PL/I does not. In the following SPL code fragment:

```
X : for (n = 0; n < 10; n++)
{
    ...
    goto X;
    ...
    break X;
    ...
    continue X;
}
```

the three statements “`goto X`”, “`break X`” and “`continue X`” within the loop body all use the same label, yet each statement causes a jump to a different location (i.e. “`goto X`” causes the entire loop to be re-started from the beginning, “`break X`” causes the loop to finish at once and “`continue X`” causes an increment of `n` by 1 and a jump to the next loop iteration).

Now that the `label` variables were a part of the language, it was only logical to allow using `label`-yielding expressions as jump targets:

```
declare x : label;
...
for (n = 0; n < 10; n++)
{
    ...
    goto x;
    ...
    break x;
}
```

```

...
    continue x;
}

```

Not only was a label variable required to remember all three possible jump locations – it could now be the case that a label variable was assigned a value that disallowed its use as a jump target in some contexts. Consider:

```

declare x : label;
...
Y : x = Y;
...
for (n = 0; n < 10; n++)
{
    ...
    goto x;
    ...
    break x;    // ERROR
    ...
    continue x; // ERROR
}

```

Although the statement “goto x” would still be valid (it would cause the transfer of control to the statement labelled by Y), the statements “break x” and “continue x” would not – as the statement labelled by Y is not a loop statement that could be exited or re-iterated.

Another potential danger lied in the fact that loop labels were now available outside of the loop scope:

```

for (m = 0; m < 10; m++)
{
    declare x : label;
    ...
    for (n = 0; n < 10; n++)
    {
        Y : x = Y;
        ...
    }
    ...
    goto x;    // ERROR
    ...
    break x;   // ERROR
    ...
    continue x; // ERROR
}

```

In the above code fragment, all three jump statements cause a jump into a loop that is no longer active.

All in all, an introduction of `label` variables into SPL was an exercise where increased orthogonality went hand-in-hand with increased margin for programmer errors, as well as potentially increased runtime cost in some of the more exotic cases. However, to a programmer coming from C (i.e. a programmer not using `label` variables or nonlocal jumps) there would be no extra penalty (and the C way of doing nonlocal jumps via `set jmp/long jmp` standard library functions is exactly as costly at runtime as SPL nonlocal jumps or, for that matter, C++ exception handling).

## Lexical namespaces

Having written a relatively large amount of SPL code, it was becoming more and more clear to me that SPL was no more suitable for modular programming than C was. Although some people may agree that that's enough, I wanted more. I wanted C++ - like namespaces (or Modula-2 – like packages. Or something similar). I wanted enumerated types to act as scopes for enumeration literals (as they do in Java or C# - neither C nor C++ have this ability, all enumeration literals in both C and C++ are global).

What I did *not* want was to introduce new concepts into the language. Having previously participated in the development of a C++ compiler I knew full well just how much effort would be required to properly implement namespaces (let alone packages, which are namespaces with visibility constraints on top).

I needed an alternative solution that would be as lightweight as possible, yet would put away name conflicts once and for all.

I was fully aware of how the name pollution problem was solved in C world – and that was by using name prefixes, as in:

```
enum x { x_a, x_b, x_c };  
int package1_function() ...  
int package2_function() ...
```

Quite apart from leading to identifiers that are both ugly and long, this approach has an even more serious problem. The problem is an existence of a legacy C naming convention originating in times when linkers were case-insensitive – under this naming convention compound identifiers (i.e. identifiers consisting of several words) are written in all-lowercase form, with underscores used as word separators. This is still the preferred naming convention in GNU/Linux community, even though the original reason for all-lowercase names (namely, the linker case-insensitivity) is long gone. Nevertheless, people are people, and what is `numberOfClients` to me will forever be `number_of_clients` to them. The problem starts when an identifier such as `package1_number_of_clients` appears in the source code – as it is now impossible to tell where the package portion ends.

I quite liked the C++ form (where the above identifier would be written as either `package1::numberOfClients` or `package1::number_of_clients`, depending on the naming convention in use). In fact, I would have been happier yet if the same convention were to apply to C++ enumeration literals (i.e. after a type

declaration `enum x { a, b, c }` one had to write `x::a` to refer to an enumeration literal).

And then it hit me – why not allow double-colons within identifiers, treating double-colon as a special compound letter? This will allow SPL entities to have identifiers that *look* like they belong to a specific namespace, while they all reside in the same global namespace:

```
type Radix = (Radix::Binary, Radix::Octal,
             Radix::Decimal, Radix::Hexadecimal);
declare r : Radix = Radix::Binary;
...
declare package1::numberOfClients : ...
declare package2::numberOfClients : ...
...
procedure package1::someFunction() ...
procedure package2::someFunction() ...
```

Unlike C++, where double-colon `::` is a separate token, in SPL it is an integral part of an identifier. A minor consequence is in that C++ allows spaces around double-colons in compound identifiers while SPL does not; however, I have never seen, let alone written, a piece of C++ code where `package1::numberOfClients` is written as `package1 :: numberOfClients`. As SPL does not have any context where two consecutive colons are allowed, no ambiguity was introduced into the language.

While allowing double-colons in identifiers effectively solved all name pollution problems, I now had another problem on my hands – these identifiers were long. *Very* long. Too long, even for me. With proper namespaces, I would have used something like C++ `using` directive (which bring namespace members into the current scope, where they can be referred to by their unqualified names), but what I had in SPL was not namespaces.

However, it was a form similar to the C++ `using` directive that eventually made its way into SPL. Whereas in C++ the directive “`using X`” means “bring all members of namespace `X` into the current scope”, its SPL counterpart “`using X`” means “when resolving some name `Y` in the current scope, try `X::Y` as well as `Y` and see what happens”.

On the surface, SPL code looks very much like the corresponding C++ code:

C++	SPL
<pre>namespace package1 {     int numberOfClients; } ... using package1; ...</pre>	<pre>declare     package1::numberOfClients :         integer; ... using package1; ... numberOfClients++;</pre>

<code>numberOfClients++;</code>
---------------------------------

Under the hood, the things work quite differently. In C++ case, the `using` directive brings all members of the `package1` namespace into the current scope, so the statement “`numberOfClients++`” resolves the identifier `numberOfClients` to the namespace member. In SPL case, the `using` directive says “try prefixing all identifiers with `package1::` before looking them up”, so when an identifier `numberOfClients` is encountered the compiler looks up two identifiers – `numberOfClients` and `package1::numberOfClients`. Since only one of them is found, the lookup is unambiguous.

To this day, I believe the introduction of what I call “lexical namespaces” (because things look like namespaces, yet are resolved on a lexical level, by identifier prefixing) to be the ultimate achievement in the SPL design – a very low-level feature dressed up in a very high-level syntax (which is what SPL is all about).

To tell the story in full, the only reason I was, at first, somewhat reserved about allowing double-colons within identifiers was in that I still planned to produce further SPL-based languages with named namespaces and user-defined classes (that’s SPL+ and SPL++, mentioned at the beginning of this document), and I knew that allowing double-colons within identifiers will make it impossible to use these double-colons as compound name separators (as C++ does). Finally, I have decided to let things go, with an explicit understanding that a dot ‘.’ symbol will be used to denote a membership of an entity within a namespace or class – as it does in Modula-2, Java and C#.

## Opaque types

As the SPL compiler sources grew in size, I wanted to modularize them. Even more importantly, now that major SPL features were sorted out, I wanted to provide some support for data encapsulation – quite apart from all the SPL code that could be written in the future, this was needed for the SPL standard library.

From my C and C++ experience I was well familiar with the trick where an identifier is declared to be a structure (or a class), but no actual definition for the structure (or class) is given as in:

```
struct X;
// We can use X*, even though X is an incomplete type
X * foo();
void bar(X * pX);
```

So far so good – except in the above C code fragment the fact that `X` is a structure is still visible, even though the contents of the structure is not.

What I needed was a way to declare that some identifier refers to a type without saying what kind of type it is. The type declaration syntax was easy enough to adopt for this role:

```

type X;
// We can use ^X, even though X is an opaque type
procedure foo() : ^X;
procedure bar(pX : ^X);

```

The idea was in that a header would declare an opaque type along with a set of procedures that operate on instances of this type (such as the code snippet above). The implementation file will `#include` the header, but will also give a complete definition for the opaque types, as well as provide procedure bodies. In this way, the actual representation of an opaque type will be encapsulated within a single compilation unit.

Naturally, using the above approach it is not possible to manipulate instances of opaque types (because, among other things, the size of these instances is not known); however, it is possible to manipulate pointers to these instances (the same constraint applies to Modula-2 opaque types, much for the same reason). Fortunately, it is always possible to name the pointer type:

```

type StackImpl; // Opaque
type Stack = ^StackImpl;

procedure Stack::create() : Stack;
procedure Stack::destroy(stack : Stack);
...

```

Note that procedures operating on `Stack` instances are assigned qualified names which, to the user of the stack module above, look like they are scoped to the `Stack` types.

I was fully aware that this approach to opaque types (also found in Modula-2) was historically regarded as not-quite-efficient. The most serious concern was usually about very small procedures operating on opaque type instances (such as returning one of their fields) being translated into full-blown procedure calls (as opposed to C++ inline class functions, that provide the needed security without the need for an actual procedure call). However, I saw the inefficiency of opaque types as a language implementation problem rather than something inherent in the language – my experience with link-time inlining has taught me that it is possible to avoid most of the actual procedure calls, whereas link-time code generation would result in as efficient a code as if the opaque type was not encapsulated at all.

In a way, SPL provides better encapsulation facilities than C++ – a C++ class declaration may limit the access to class members, yet it still contains declarations of all these class members, whether accessible or not. A user of such a class may not be able to *do* anything with its private members, but he can still *see* what these members are. As C++ class libraries have to be distributed along with the appropriate headers, at least part of their implementation is explicitly present in these headers. In SPL, on the other hand, the only part of an ADT that appears in the headers is its name (to be precise, the same programming technique *could* be used in both C and C++, except C programmers usually don't bother to encapsulate, whereas C++ programmers tend to

rely on class member access control instead of defining proper opaque types – whereas an SPL programmer has no choice but to do the things properly).

## Anonymous fields

Anonymous fields were another feature of C which was picked up, although this happened quite late in SPL lifetime.

The main reason for including anonymous fields into the language was the need to model type hierarchies within the SPL compiler. To give but one example, consider an internal representation of types within the compiler – all types have a common set of fields, but some of them need an extra:

```
type GIR_TYPE = new structure
{
    category                : GIR_TYPE_CATEGORY;
    size                    : cardinal*8;
    alignment                : cardinal;
    ... // More fields
};

type GIR_POINTER_TO_TYPE = new structure
{
    category                : GIR_TYPE_CATEGORY;
    size                    : cardinal*8;
    alignment                : cardinal;
    ... // More GIR_TYPE fields
    pBaseType                : ^GIR_TYPE;
};
```

Repeating declarations common for all types over and over again, once for each subtype of GIR\_TYPE was tiring to say the least. Eventually I have made some use of the SPL preprocessor to reduce the amount of code I was typing in, but I knew full well it was a temporary measure.

Normally, I would have used sub-objects of base types as derived type fields, as in:

```
type GIR_POINTER_TO_TYPE = new structure
{
    base                    : GIR_TYPE;
    pBaseType                : ^GIR_TYPE;
};
```

except this would automatically mean an extra member in the type qualification chain:

```
declare pT : ^GIR_TYPE;
declare pPT : ^GIR_POINTER_TO_TYPE;
...
if (pT->alignment == 1u) ...
if (pPT->base.alignment == 1u) ...
```

The solution to this mess was not apparent until, at one time, I have encountered a bug in a C++ compiler I was working on independently for a different customer, the bug being related to an improper treatment of anonymous unions. I did not have a lot of experience with anonymous unions before that time; however, the need to fix a C++ compiler bug caused me to open the ISO C++ language specification and read the chapter on anonymous unions. At that very moment, I knew SPL needed them too.

Inclusion of anonymous unions into SPL went surprisingly smoothly, although, for the sake of orthogonality, they have become anonymous fields (i.e. a structure or union can have unnamed fields of any type, not just of a union type). With this feature, the above code became:

```
type GIR_TYPE = new structure
{
    category                : GIR_TYPE_CATEGORY;
    size                    : cardinal*8;
    alignment                : cardinal;
    ... // More fields
};
```

```
type GIR_POINTER_TO_TYPE = new structure
{
    *                       : GIR_TYPE;
    pBaseType                : ^GIR_TYPE;
};
```

```
...
declare pT : ^GIR_TYPE;
declare pPT : ^GIR_POINTER_TO_TYPE;
...
if (pT->alignment == 1u) ...
if (pPT->alignment == 1u) ...
```

Naturally, anonymous fields of types other than structure or union cannot be accessed directly (as they cannot be further qualified); however, I have soon found a way for them to be useful – namely in acting as padding when a precise structure layout is important:

```
type T = packed structure
{
    c : character*1;           // 1 byte
    * : array [3] of cardinal*1; // 3 bytes of padding
    n : cardinal*4;           // 4 more bytes
};
```

## Extending the preprocessor

Being modelled after the C preprocessor, the SPL preprocessor at the time already had the `#error` directive, which could be used to issue user-defined errors during compilation. I do not remember which C toolchain I saw a corresponding `#warning` directive in (for issuing user-defined warnings, it may have been GCC), but see it I did, and, having judged it potentially useful, adding it to the SPL preprocessor was a

matter of minutes. The same went for a `#message` directive, which issues a user-defined message that is neither a warning nor an error.

One more preprocessor directive was caused by a somewhat frequent need to ensure preprocessing consistency. From my C and C++ experience I remembered just how frequently I had to write something along the lines of:

```
#if !<some condition>
    #error <some error message>
#endif
```

In the SPL language itself (as opposed to SPL preprocessor), such situations are handled with an `assert` statement, so I saw a natural parallel there and decided to introduce an `#assert` preprocessor directive:

```
#assert <some condition>
```

which would cause an error message if the specified condition evaluated to `false`.

To give the full account of the story, I have later discovered an `#assert` directive in AIX C/C++ compiler by IBM, which had a completely different meaning there. Needless to say, this discovery had no effect on the SPL preprocessor.

At about the same time, I saw C99 specification for variadic macros and decided it was a good idea to have them. Initially the placeholder for variadic arguments was called `VA_ARGS`, which was soon changed to `__VA_ARGS__` to keep SPL preprocessor closer to C99 preprocessor.

## Rewriting the compiler

Now that I knew what new features I wanted in the language, the SPL compiler was rewritten from scratch, thus making it an increment 3. Although I could not use newer SPL features (such as opaque types or lexical scoping), the increment 3 of an SPL compiler was written in a very modular manner, with a lot of comments along the lines of “in proper SPL, this should be reprogrammed using (for example) anonymous fields (or call-by-constant reference, or as something else)”. The intent of such comment was to help when the SPL compiler is rewritten once again in the “proper” SPL, what with all the new features.

The main difference between increments 2 and 3 of the SPL compiler was in that while increment 2 performed parsing and semantic processing all in one pass (resulting in a graph-based program IR that was then passed on to the code generator), increment 3 did only the parsing during the first pass, obtaining the abstract syntax tree of the program. After that, a total of 13 passes over the AST were employed to build the graph-based program IR suitable for code generation. The number may seem excessive and, with some thought, it could probably be made far lower; however, I wanted the IR building routines to be as simple as possible (so that they would be easier to debug), so each of these 13 passes performed only one AST → IR transformation task (for example, pass #1 was concerned *only* about what scopes there existed and what identifiers were declared in each scope, pass #2 was concerned *only*

about enumeration literals, etc.) All in all, even with 14-pass analysis the increment 3 of the SPL compiler was comparable both in speed and resource usage to increment 2, which was all I needed.

One of the new features of the increment 3 of the SPL compiler was its ability to write program AST, program IR and entity dependency graph into XML text files. This ability was initially intended to allow interfacing SPL compiler with tools such as program analyzers and code browsers; however, it have eventually proven invaluable in tracking the more subtle bugs within the SPL compiler itself. Later on, I was quite surprised to find that, when SPL documentation became publicly downloadable, the specifications of these XML DTDs were downloaded almost twice as frequently as the SPL language reference.

# Rewriting the standard library

Now that the new SPL facilities were at my disposal, I seriously started to think about extending the standard library.

## Extensibility

One of the problems with the SPL standard library that I wanted solved was its extensibility. The SPL standard library provided a number of stock items to a programmer (such as error codes, character sets, language and country codes, format specifications, etc.) – the only problem was that this set was fixed. My previous experience with C has taught me time and time again just how frustrating that can be.

To give but one example, the C standard library provides a global variable `errno`, which is assigned an error code when one of the standard library services cannot succeed; in addition, a number of symbolic constants is defined for some specific error codes, as well as a `strerror()` service, which yields a printable error message given an error code as a parameter.

So far so good. The bad things start when I need a custom error code – in one of the C++ projects I did at the time it was necessary to report “normal” I/O errors as well as user-defined I/O errors (the project was an ELF object module handling library, where “normal” I/O errors, like “file does not exist” or “disk is full”, could occur just as likely as ELF-specific I/O errors, such as “section extends beyond object file end”). It was quite easy to define additional error codes for the user-defined errors I needed, but it was impossible to teach `strerror()` to yield proper error messages for these custom error codes.

Another example is the formatted I/O. C defines a `printf/scanf` family of routines which do a pretty good job of formatted I/O for all of C primitive types. Indeed, the `printf/scanf` family of routines is so convenient that a large amount of C++ code exists that uses `printf/scanf` routines instead of the C++ alternative (text streams).

Yet again, so far so good. The problems arise when I want to, for example, print a value of a user-defined type. As the set of supported format specifiers is fixed, the task is impossible, forcing the programmer to break printing into primitive portions. For example, given the definitions:

```
type Point = structure { x, y : integer; };  
declare p : Point;
```

the only way to print values of type `Point` is by printing its components individually, as in:

```
Printf("( %d%, %d%)", p.x, p.y);
```

(Note that the SPL `Printf/Scanf` family of routines have originally required percent signs both before and after a format specification, hence what was `%d` in C

became `%d` in SPL. This is due to the fact that SPL provides platform-independent primitive types, so `%d` meant “print an integer argument” whereas `%d8` meant “print an integer\*8 argument”, and so on).

The need to allow user to extend the SPL standard library led to a design where for all stock entities defined by the SPL standard library (that includes error codes, format specifications, character sets, time zones, etc.) the standard library provides some predefined instances – but it also provides a way to register new error codes, format specifications, character sets, time zones, etc. Taking format specifications as an example, the user could use some predefined format specifications to print values of primitive SPL types, as in:

```
declare n : integer;
...
Printf("{integer}", n);
```

(note that the syntax of format specifications has changed to using curly braces instead of percent signs for formatting placeholders); however, a user is also given an ability to define his own format specifications, which allows code like:

```
declare p : Point;
...
Printf("{Point}", p);
```

to be written. Similarly, user can define his own character sets, time zones, languages, countries, error codes – any set of stock entities provided by the SPL standard libraries could now be extended if necessary.

## Scoping the library

The original SPL standard library was similar to C standard library in that everything it provided existed in the same global namespace. However, I now had lexical scoping at my disposal, so the first thing to do was to prefix all standard library services with “`spl::`”, as in:

```
declare n : integer;
...
spl::Printf("{integer}", n);
```

Although this could lead to a programmer having to write longer identifiers, the problem was easily solvable with the `using` directive:

```
using spl;
...
declare n : integer;
...
Printf("{integer}", n);
```

After that, a long and serious deliberation occurred.

With lexical scoping at my fingertips, I could (and, moreover, I *wanted* to) reorganize the entire standard SPL standard library interface to look like an OO API, along the following lines:

```
// Somewhere in SPL standard header io.h ...
type spl::File = ...;
constant spl::File::None = ...;
procedure spl::File::open(...) ...;
procedure spl::File::close(f : spl::File) ...;
procedure spl::File::read(f : spl::File, ...) ...;

// Somewhere else, in user code ...
using spl;
...
declare f : File = File::open(...);
if (f <> File::None)
{
    ...
    File::read(f, ...);
    ...
    File::close(f);
}
```

Although the resulting SPL code will, in my opinion, become far more readable and maintainable, it would not be POSIX-compliant, which would force programmers with C or C++ background to learn a whole new standard library API. I was not comfortable with the implications – until one fine day I have realized that there’s nothing in the world that would prevent me from providing *both* the OO-like SPL standard library *and* a POSIX emulation layer on top of that. With lexical scoping, I could scope all POSIX stuff into a separate lexical namespace, such as:

```
// Somewhere in a POSIX header ...
type posix::char = character*1;
type posix::int = integer*4;
...
procedure posix::atoi(
    s : ^constant posix::char
) : posix::int;

// Somewhere else, in user code ...
using posix;
...
declare n : int;
declare s : array [100] of char;
...
n = atoi(s);
```

Once POSIX compatibility was out of the way, SPL standard library facilities were quickly assigned to a number of ADTs and their names changed appropriately (for example, formatted output functions `Printf`, `CPrintf` and `FPrintf`, which send

formatted output to console, standard output stream and an arbitrary file respectively, have become `spl::Stdout::print`, `spl::Console::print` and `spl::File::print`, correspondingly, with `spl::Stderr::print` thrown in for completeness). An extra benefit was an ability to scope symbolic constants to their types:

```
type spl::File::ShareMode = new cardinal*4;
constant spl::File::ShareMode::None = ...;
constant spl::File::ShareMode::Read = ...;
constant spl::File::ShareMode::Write = ...;
...
type spl::impl::LocaleImpl; // Opaque
type spl::Locale = ^spl::impl::LocaleImpl;
constant spl::Locale::None = spl::Locale(0);
constant spl::Locale::System = spl::Locale(1);
constant spl::Locale::Default = spl::Locale(2);
...
// And so on ...
```

## Environments

The distinction between hosted and freestanding environment is another one of the C standard library features that has been adopted by the rewritten SPL standard library.

While the 1<sup>st</sup> increment of the SPL standard library was written mainly to facilitate porting SPL compiler to SPL, the 2<sup>nd</sup> increment was now being written as more-or-less final version, which would be rolled out along with the compiler. This being the case, I had to think about how the SPL standard library will be used for what was SPL primary design niche – writing OS kernels. The idea of the hosted vs freestanding environment distinction fit in almost perfectly.

In its basics, the hosted environment means full SPL standard library, while freestanding environment means a subset of the SPL standard library that does not require OS support. As part of enforcing this distinction, all OS-dependent declaration in the SPL standard library headers were now guarded with conditional compilation directives, making the non-freestanding parts of the standard library absent when compiling for a freestanding target.

As a note on SPL in a hosted environment, the decision to keep program entry point separate from command line arguments played out well. In C, a program written for a hosted environment always starts with the function `main()`, which also accepts command-line arguments:

```
int main(int argc, char ** argv) ...
```

The problem with this approach is in that a different entry point is needed if we want to access 16- or 32-bit command line. Different toolchains solve this problem differently; for example, in a MSVC++ .NET toolchain a programmer can define an alternative entry point:

```
int wmain(int argc, wchar_t ** argv) ...
```

if a Unicode BMP command line needs to be accessed. Quite apart from introducing unnecessary complications (like: what to do if both `main` and `wmain` are defined?), this will never work in a language with 4 different character types. Therefore, SPL mandates that in a hosted environment the procedure `main()` without any parameters always acts as a program entry point, and that command-line arguments are accessed using dedicated command line API provided by the standard library, as in:

```
#include "cmdline.sph" from SPL_STANDARD_LIBRARY_HEADERS
using spl;

procedure main() public
{
    for (declare i : cardinal = 0u;
        i < CommandLine::getArgumentCount();
        i++)
    {
        declare arg : ^constant character =
            CommandLine::getArgument(i);
        // Do something with arg...
    }
}
```

or, if an ISO-10646 command line is needed:

```
#include "cmdline.sph" from SPL_STANDARD_LIBRARY_HEADERS
using spl;

procedure main() public
{
    for (declare i : cardinal = 0u;
        i < CommandLine::getArgumentCount();
        i++)
    {
        declare arg : ^constant character*4 =
            CommandLine::getArgumentS4(i);
        // Do something with arg...
    }
}
```

## Generic containers

While being modelled after the C standard library (even though library facilities were now named differently), SPL standard library was extended to include something the C standard library never had – namely generic containers.

The decision to include generic containers into the SPL standard library was geared mainly by the fact that the SPL compiler needed dynamic arrays, string buffers, hash tables and sets over and over again – within increment 3 of an SPL compiler all these features were provided by two dedicated components (one for string buffer handling, one for generic containers). For all these features to be moved to the SPL standard

library meant that the SPL compiler source would become that much smaller in the next increment – but, even more importantly, it meant that SPL users will not have to waste time writing their own code for the same purpose. Being at the time involved in the development of an independent C/C++ toolchain, I saw generic containers being reused all over the code, which meant to me that it was a good idea to provide them upfront.

As I see it now, this was the first step that SPL standard library took from being feature-equivalent to C standard library towards being feature-equivalent to C++ standard library. Further inclusion of variable-length string buffer services into SPL standard library was just the next step in the same directions; other steps followed promptly.

# Coming of age

Yet to be written...

# Appendix A: GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in

part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## **2. VERBATIM COPYING**

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## **3. COPYING IN QUANTITY**

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### **4. MODIFICATIONS**

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## **5. COMBINING DOCUMENTS**

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

## **6. COLLECTIONS OF DOCUMENTS**

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## **7. AGGREGATION WITH INDEPENDENT WORKS**

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## **8. TRANSLATION**

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## **9. TERMINATION**

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## **10. FUTURE REVISIONS OF THIS LICENSE**

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

# Appendix B: SPL grammar (increment 1)

This appendix contains the SPL grammar as implemented by increment 1 of the SPL compiler in early 2003.

## Notation

The following notation is used in describing the SPL grammar:

- <X> – Non-terminal symbol X.
- x – Terminal symbol x.
- x | y – Alternative, either x or y.
- [x] – Optionality, x can appear once or it can not appear at all.
- [x ...] – Repetition, x can appear zero or more times.

## Comments

SPL recognized C++ style comments, i.e.:

- Anything between /\* and \*/ is a comment. Such comments can span multiple lines and cannot be nested.
- Anything between the // and the following end-of-line is a comment.

## Lexical elements

<LOWER>	::= <u>a</u>   <u>b</u>   <u>c</u>   <u>d</u>   <u>e</u>   <u>f</u>   <u>g</u>   <u>h</u>   <u>i</u>   <u>j</u>   <u>k</u>   <u>l</u>   <u>m</u>   <u>n</u>   <u>o</u>   <u>p</u>   <u>q</u>   <u>r</u>   <u>s</u>   <u>t</u>   <u>u</u>   <u>v</u>   <u>w</u>   <u>x</u>   <u>y</u>   <u>z</u>
<UPPER>	::= <u>A</u>   <u>B</u>   <u>C</u>   <u>D</u>   <u>E</u>   <u>F</u>   <u>G</u>   <u>H</u>   <u>I</u>   <u>J</u>   <u>K</u>   <u>L</u>   <u>M</u>   <u>N</u>   <u>O</u>   <u>P</u>   <u>Q</u>   <u>R</u>   <u>S</u>   <u>T</u>   <u>U</u>   <u>V</u>   <u>W</u>   <u>X</u>   <u>Y</u>   <u>Z</u>
<DIGIT>	::= <u>0</u>   <u>1</u>   <u>2</u>   <u>3</u>   <u>4</u>   <u>5</u>   <u>6</u>   <u>7</u>   <u>8</u>   <u>9</u>
<DIGIT1>	::= <u>1</u>   <u>2</u>   <u>3</u>   <u>4</u>   <u>5</u>   <u>6</u>   <u>7</u>   <u>8</u>   <u>9</u>
<ODIGIT>	::= <u>0</u>   <u>1</u>   <u>2</u>   <u>3</u>   <u>4</u>   <u>5</u>   <u>6</u>   <u>7</u>
<XDIGIT>	::= <DIGIT>   <u>a</u>   <u>b</u>   <u>c</u>   <u>d</u>   <u>e</u>   <u>f</u>   <u>A</u>   <u>B</u>   <u>C</u>   <u>D</u>   <u>E</u>   <u>F</u>
<NSPEC>	::= <u>_</u>   <u>\$</u>   <u>@</u>
<ALPHA>	::= <LOWER>   <UPPER>
<ALNUM>	::= <ALPHA>   <DIGIT>
<NSYMF>	::= <ALPHA>   <NSPEC>
<NSYM>	::= <ALNUM>   <NSPEC>
<NAME>	::= <NSYMF> [ <NSYM> ... ]
<X>	::= <u>x</u>   <u>X</u>



```

<STRING8> ::= [ <A> ] _ [ <CHAR8> ... ] _
<STRING16> ::= <U> _ [ <CHAR16> ... ] _
<STRING32> ::= <I> _ [ <CHAR32> ... ] _
<A> ::= a | A
<U> ::= u | U
<I> ::= i | I
<CHARACTER> ::= <CHARACTER8> | <CHARACTER16>
                | <CHARACTER32>
<STRING> ::= <STRING8> [ <STRING8> ... ]
                | <STRING16> [ <STRING16> ... ]
                | <STRING32> [ <STRING32> ... ]

```

## LL(k) syntax

```

<translation unit> ::= [ <declaration> ... ]
<declaration> ::= <type declaration>
                | <constant declaration>
                | <variable declaration>
                | <procedure declaration>
<type declaration> ::= type <typedef> [ , <typedef> ... ] ;
<typedef> ::= <NAME> [ is <type attribute>
                    [ <type attribute> ... ] ]
<type attribute> ::= <type signature>
                  | new
<constant declaration> ::= constant <constdef> [ , <constdef> ... ] ;
<constdef> ::= <NAME> is <el>
<variable declaration> ::= declare <vardef> [ , <vardef> ... ] ;
<vardef> ::= <name list> as
            [ <variable attribute> ... ]
<name list> ::= <NAME> [ , <NAME> ... ]
<variable attribute> ::= <type signature>
                    | <memory class>
                    | <visibility>
                    | <alias>
                    | <segment>
                    | <global>
                    | <initializer>
<memory class> ::= static
                  | automatic
                  | register
                  | shared
                  | thread

```

```

<visibility> ::= public
              | private
              | external
              | export
              | import <STRING>

<alias> ::= alias <STRING>

<segment> ::= segment <STRING>

<global> ::= global <STRING>

<initializer> ::= initial <e1>

<procedure declaration> ::= <procedure header>
                           <procedure body>

<procedure header> ::= procedure <NAME>
                      [ <parameter list> ] <return type>
                      [ <procedure attribute> ...]

<parameter list> ::= ( [ <parameters> ] )

<parameters> ::= <parameter> [ , <parameter> ...] [ , ... ]
                | ...

<parameter> ::= <optname list> :
                [ <parameter attribute> ...]

<optname list> ::= <optname> [ , <optname> ...]

<optname> ::= <NAME> | *

<parameter attribute> ::= <type signature>
                          | <memory class>
                          | <parameter direction>

<parameter direction> ::= in | out | in out

<return type> ::= [ : <type signature> ]

<procedure attribute> ::= <visibility>
                          | <alias>
                          | <global>
                          | <segment>
                          | inline

<procedure body> ::= i
                    | [ <condition> ...]
                    <block statement>

<condition> ::= <precondition>
                | <postcondition>

<precondition> ::= [ <NAME> : ] require <expression> i

<postcondition> ::= [ <NAME> : ] ensure <expression> i

<type signature> ::= <primitive type>
                   | <named type>

```

```

| <pointer type>
| <structure type>
| <union type>
| <enumerated type>
| <array type>
| <procedure type>

<primitive type> ::= integer*1
| integer*2
| integer*4
| integer*8
| integer
| cardinal*1
| cardinal*2
| cardinal*4
| cardinal*8
| cardinal
| real*4
| real*8
| real
| character*1
| character*2
| character*4
| character
| pointer
| boolean

<named type> ::= <NAME>

<pointer type> ::= ^ <type signature>

<structure type> ::= structure <fields> end

<union type> ::= union <fields> end

<enumerated type> ::= ( [ <enumeration literal list> ] )

<enumeration literal list> ::= <enumeration literal>
[ , <enumeration literal> ...]

<enumeration literal> ::= <NAME>

<array type> ::= array <dimensions> of <type signature>

<dimensions> ::= [ <dimension> ] [ [ <dimension> ] ...]

<dimension> ::= <e1> | *

<fields> ::= [ <field> ...]

<field> ::= <optname list> : <type signature> ;

<procedure type> ::= procedure <parameter list> <return type>

<ds> ::= <statement>
| <declaration>

<statement> ::= <NAME> : <statement>
| <empty statement>
| <expression statement>
| <block statement>

```

		<if statement>
		<switch statement>
		<while statement>
		<do statement>
		<for statement>
		<goto statement>
		<break statement>
		<continue statement>
		<return statement>
		<assert statement>
		<delete statement>
		<with statement>
<empty statement>	::=	<u>i</u>
<expression statement>	::=	<expression> <u>i</u>
<block statement>	::=	<u>begin</u> [ <ds> ] <u>end i</u>
<if statement>	::=	<u>if</u> <expression> <u>then</u> <statement> [ <u>else</u> <statement> ]
<switch statement>	::=	<u>switch</u> <expression> <u>is</u> { [ <alternative> ...] }
<alternative>	::=	<selector> [ <selector> ...] <ds> [ <ds> ...]
<selector>	::=	<u>when</u> <expression> [ <u>...</u> <expression> ] <u>:</u>   <u>default</u> <u>:</u>
<while statement>	::=	<u>while</u> <expression> <u>do</u> <statement>
<do statement>	::=	<u>do</u> <statement> <u>while</u> <expression> <u>i</u>
<for statement>	::=	<u>for</u> ( [ <expression> ] <u>i</u> [ <expression> ] <u>i</u> [ <expression> ] ) <statement>
<goto statement>	::=	<u>goto</u> <NAME> <u>i</u>
<break statement>	::=	<u>break</u> [ <NAME> ] <u>i</u>
<continue statement>	::=	<u>continue</u> [ <NAME> ] <u>i</u>
<return statement>	::=	<u>return</u> [ <expression> ] <u>i</u>
<assert statement>	::=	<u>assert</u> <expression> <u>i</u>
<delete statement>	::=	<u>delete</u> <expression> <u>i</u>
<with statement>	::=	<u>with</u> <expression> <u>do</u> <statement>
<expression>	::=	<e1> [ <u>,</u> <e1> ...]
<e1>	::=	<e2> [ <assign op> <e1> ]

```

<assign op> ::= =
              | <binary op> =

<binary op> ::= + | - | * | / | %
              | << | >>
              | == | <> | < | <= | > | >=
              | and [ then ]
              | implies [ then ]
              | xor
              | or [ else ]

<e2> ::= <e3> [ ? <e2> : <e2> ]

<e3> ::= <e4> [ <or op> <e4> ...]

<or op> ::= or [ else ]

<e4> ::= <e5> [ xor <e5> ...]

<e5> ::= <e6> [ <implies op> <e6> ...]

<implies op> ::= implies [ then ]

<e6> ::= <e7> [ <and op> <e7> ...]

<and op> ::= and [ then ]

<e7> ::= <e8> [ <cmp op> <e8> ...]

<cmp op> ::= == | <> | < | <= | > | >=

<e8> ::= <e9> [ <shift op> <e9> ...]

<shift op> ::= << | >>

<e9> ::= <e10> [ <add op> <e10> ...]

<add op> ::= + | -

<e10> ::= <e11> [ <mul op> <e11> ...]

<mul op> ::= * | / | %

<e11> ::= <un op> <e11>
              | <e12>

<un op> ::= + | - | not | & | ++ | --

<e12> ::= <e13> [ <e12suffix> ...]

<e12suffix> ::= . <NAME>
              | -> <NAME>
              | ^
              | [ <expression> ]
              | ( [ <e1> [ , <e1> ...] ] )
              | ++
              | --

<e13> ::= ( <expression> )
              | <NAME>
              | <INTEGER>

```

```
| <REAL>  
| <CHARACTER>  
| <STRING>  
| true  
| false  
| null  
| result  
| <type signature> ( <e1> [ , <e1> ... ] )  
| sizeof ( <expression> )  
| sizeof ( <type signature> )  
| new <type signature> [ [ <expression> ] ]  
| . <NAME>  
| -> <NAME>
```

## Appendix C: SPL grammar (increment 2)

This appendix contains the SPL grammar as implemented by increment 2 of the SPL compiler towards mid-2005.

### Notation

The following notation is used in describing the SPL grammar:

<X>	– Non-terminal symbol X.
<u>x</u>	– Terminal symbol x.
x   y	– Alternative, either x or y.
[ x ]	– Optionality, x can appear once or it can not appear at all.
[ x ... ]	– Repetition, x can appear zero or more times.

### Comments

SPL recognized C++ style comments, i.e.:

- Anything between /\* and \*/ is a comment. Such comments can span multiple lines and cannot be nested.
- Anything between the // and the following end-of-line is a comment.

### Lexical elements

<LOWER>	::= <u>a</u>   <u>b</u>   <u>c</u>   <u>d</u>   <u>e</u>   <u>f</u>   <u>g</u>   <u>h</u>   <u>i</u>   <u>j</u>   <u>k</u>   <u>l</u>   <u>m</u>   <u>n</u>   <u>o</u>   <u>p</u>   <u>q</u>   <u>r</u>   <u>s</u>   <u>t</u>   <u>u</u>   <u>v</u>   <u>w</u>   <u>x</u>   <u>y</u>   <u>z</u>
<UPPER>	::= <u>A</u>   <u>B</u>   <u>C</u>   <u>D</u>   <u>E</u>   <u>F</u>   <u>G</u>   <u>H</u>   <u>I</u>   <u>J</u>   <u>K</u>   <u>L</u>   <u>M</u>   <u>N</u>   <u>O</u>   <u>P</u>   <u>Q</u>   <u>R</u>   <u>S</u>   <u>T</u>   <u>U</u>   <u>V</u>   <u>W</u>   <u>X</u>   <u>Y</u>   <u>Z</u>
<DIGIT>	::= <u>0</u>   <u>1</u>   <u>2</u>   <u>3</u>   <u>4</u>   <u>5</u>   <u>6</u>   <u>7</u>   <u>8</u>   <u>9</u>
<DIGIT1>	::= <u>1</u>   <u>2</u>   <u>3</u>   <u>4</u>   <u>5</u>   <u>6</u>   <u>7</u>   <u>8</u>   <u>9</u>
<ODIGIT>	::= <u>0</u>   <u>1</u>   <u>2</u>   <u>3</u>   <u>4</u>   <u>5</u>   <u>6</u>   <u>7</u>
<XDIGIT>	::= <DIGIT>   <u>a</u>   <u>b</u>   <u>c</u>   <u>d</u>   <u>e</u>   <u>f</u>   <u>A</u>   <u>B</u>   <u>C</u>   <u>D</u>   <u>E</u>   <u>F</u>
<NSPEC>	::= <u>_</u>   <u>\$</u>   <u>@</u>   <u>::</u>
<ALPHA>	::= <LOWER>   <UPPER>
<ALNUM>	::= <ALPHA>   <DIGIT>
<NSYMF>	::= <ALPHA>   <NSPEC>
<NSYM>	::= <ALNUM>   <NSPEC>
<NAME>	::= <NSYMF> [ <NSYM> ... ]
<X>	::= <u>x</u>   <u>X</u>



```

<STRING8> ::= [ <A> ] _ [ <CHAR8> ... ] _
<STRING16> ::= <U> _ [ <CHAR16> ... ] _
<STRING32> ::= <I> _ [ <CHAR32> ... ] _
<A> ::= a | A
<U> ::= u | U
<I> ::= i | I
<CHARACTER> ::= <CHARACTER8> | <CHARACTER16>
                | <CHARACTER32>
<STRING> ::= <STRING8> [ <STRING8> ... ]
                | <STRING16> [ <STRING16> ... ]
                | <STRING32> [ <STRING32> ... ]

```

## LL(k) syntax

```

<translation unit> ::= [ <declaration> ... ]
<declaration> ::= <type declaration>
                | <constant declaration>
                | <variable declaration>
                | <procedure declaration>
                | <use declaration>
<type declaration> ::= type <typedef> [ _ <typedef> ... ] ;
<typedef> ::= <NAME> [ = <type attribute>
                    [ <type attribute> ... ] ]
<type attribute> ::= <type signature>
                | new
                | deprecated
<constant declaration> ::= constant <constdef> [ _ <constdef> ... ] ;
<constdef> ::= <NAME> = <constant attribute>
                [ <constant attribute> ... ]
<constant attribute> ::= <e1>
                | deprecated
<variable declaration> ::= declare <vardef> [ _ <vardef> ... ] ;
<vardef> ::= <name list> :
                [ <variable attribute> ... ]
<name list> ::= <NAME> [ _ <NAME> ... ]
<variable attribute> ::= <type signature>
                | <memory class>
                | <visibility>
                | <alias>
                | <segment>
                | <global>

```

	<u>&lt;initializer&gt;</u>
	<u>deprecated</u>
<memory class>	::= <u>static</u>
	<u>automatic</u>
	<u>register</u>
	<u>shared</u>
	<u>thread</u>
<visibility>	::= <u>public</u>
	<u>private</u>
	<u>external</u>
	<u>export</u>
	<u>import</u> <STRING>
<alias>	::= <u>alias</u> <STRING>
<segment>	::= <u>segment</u> <STRING>
<global>	::= <u>global</u> <STRING>
<initializer>	::= <u>=</u> <e1>
<procedure declaration>	::= <procedure header>
	<procedure body>
<procedure header>	::= <u>procedure</u> <NAME>
	<parameter list> <return type>
	[ <procedure attribute> ...]
<parameter list>	::= ( [ <parameters> ] )
<parameters>	::= <parameter> [ <u>,</u> <parameter> ...] [ <u>,</u> <u>...</u> ]
	<u>...</u>
<parameter>	::= <optname list> <u>:</u>
	[ <parameter attribute> ...]
<optname list>	::= <optname> [ <u>,</u> <optname> ...]
<optname>	::= <NAME>   <u>*</u>
<parameter attribute>	::= <type signature>
	<memory class>
	<parameter direction>
	<u>deprecated</u>
<parameter direction>	::= <u>in</u>   <u>out</u>   <u>in out</u>
<return type>	::= [ <u>:</u> <type signature> ]
<procedure attribute>	::= <visibility>
	<alias>
	<global>
	<segment>
	<u>inline</u>
<procedure body>	::= <u>{</u>
	[ <condition> ...]
	<block statement>

```

<condition> ::= <precondition>
              | <postcondition>

<precondition> ::= [ <NAME> : ] require <expression> ;

<postcondition> ::= [ <NAME> : ] ensure <expression> ;

<use declaration> ::= use <STRING> [ , <STRING> ... ] ;

<type signature> ::= <primitive type>
                    | <named type>
                    | <pointer type>
                    | <structure type>
                    | <union type>
                    | <enumerated type>
                    | <array type>
                    | <procedure type>
                    | constant <type signature>
                    | volatile <type signature>
                    | packed <type signature>

<primitive type> ::= integer*1
                    | integer*2
                    | integer*4
                    | integer*8
                    | integer
                    | cardinal*1
                    | cardinal*2
                    | cardinal*4
                    | cardinal*8
                    | cardinal
                    | real*4
                    | real*8
                    | real
                    | character*1
                    | character*2
                    | character*4
                    | character
                    | pointer
                    | boolean

<named type> ::= <NAME>

<pointer type> ::= ^ <type signature>

<structure type> ::= structure <fields>

<union type> ::= union <fields>

<enumerated type> ::= ( [ <enumeration literal list> ] )

<enumeration literal list> ::= <enumeration literal>
                             [ , <enumeration literal> ... ]

<enumeration literal> ::= <NAME>

<array type> ::= array <dimensions> of <type signature>

<dimensions> ::= <dimension> [ <dimension> ... ]

<dimension> ::= [ [ <e1> ] ]

```

```

<fields> ::= { [ <field> ...] }

<field> ::= <optname list> : <type signature> ;

<procedure type> ::= procedure <parameter list> <return type>

<ds> ::= <statement>
      | <declaration>

<statement> ::= <NAME> : <statement>
              | <empty statement>
              | <expression statement>
              | <block statement>
              | <if statement>
              | <switch statement>
              | <while statement>
              | <do statement>
              | <for statement>
              | <goto statement>
              | <break statement>
              | <continue statement>
              | <return statement>
              | <assert statement>
              | <delete statement>
              | <with statement>

<empty statement> ::= ;

<expression statement> ::= <expression> ;

<block statement> ::= {
                    [ <ds> ]
                    }

<if statement> ::= if ( <expression> ) <statement>
                 [ else <statement> ]

<switch statement> ::= switch ( <expression> )
                     {
                     [ <alternative> ...]
                     }

<alternative> ::= <selector> [ <selector> ...]
                <ds> [ <ds> ...]

<selector> ::= when <expression> [ ... <expression> ] :
              | default :

<while statement> ::= while ( <expression> ) <statement>

<do statement> ::= do <statement> while ( <expression> ) ;

<for statement> ::= for ( [ <expression> ] ;
                       [ <expression> ] ; [ <expression> ] )
                   <statement>

<goto statement> ::= goto <NAME> ;

<break statement> ::= break [ <NAME> ] ;

```

```

<continue statement> ::= continue [ <NAME> ] ;
<return statement>  ::= return [ <expression> ] ;
<assert statement>  ::= assert <expression> ;
<delete statement>  ::= delete <expression> ;
<with statement>    ::= with ( <expression> ) <statement>
<expression>        ::= <e1> [ , <e1> ...]
<e1>                 ::= <e2> [ <assign op> <e1> ]
<assign op>          ::= =
                       | <binary op> =
<binary op>          ::= + | - | * | / | %
                       | << | >>
                       | == | <> | ≤ | ≤= | ≥ | ≥=
                       | and [ then ]
                       | implies [ then ]
                       | xor
                       | or [ else ]
<e2>                 ::= <e3> [ ? <e2> : <e2> ]
<e3>                 ::= <e4> [ <or op> <e4> ...]
<or op>              ::= or [ else ]
<e4>                 ::= <e5> [ xor <e5> ...]
<e5>                 ::= <e6> [ <implies op> <e6> ...]
<implies op>         ::= implies [ then ]
<e6>                 ::= <e7> [ <and op> <e7> ...]
<and op>             ::= and [ then ]
<e7>                 ::= <e8> [ <cmp op> <e8> ...]
<cmp op>             ::= == | <> | ≤ | ≤= | ≥ | ≥=
<e8>                 ::= <e9> [ <shift op> <e9> ...]
<shift op>          ::= << | >>
<e9>                 ::= <e10> [ <add op> <e10> ...]
<add op>             ::= + | -
<e10>                ::= <e11> [ <mul op> <e11> ...]
<mul op>             ::= * | / | %
<e11>                ::= <un op> <e11>
                       | <e12>
<un op>              ::= + | - | not | & | ++ | --

```

```

<e12> ::= <e13> [ <e12suffix> ...]

<e12suffix> ::= . <NAME>
              | -> <NAME>
              | ^
              | [ <expression> ]
              | ( [ <e1> [ . <e1> ...] ] )
              | ++
              | --

<e13> ::= ( <expression> )
        | <NAME>
        | <INTEGER>
        | <REAL>
        | <CHARACTER>
        | <STRING>
        | true
        | false
        | null
        | result
        | <type signature> ( <e1> [ . <e1> ...] )
        | sizeof ( <expression> )
        | sizeof ( <type signature> )
        | new <type signature> [ [ <expression> ] ]
        | . <NAME>
        | -> <NAME>

```