The Design and Implementation of SPECS: An Alternative C++ Syntax

Ben Werther & Damian Conway

Department of Computer Science, Monash University
Clayton, Victoria 3168, Australia

Abstract

We describe an alternative syntactic binding for C++. This new binding greatly simplifies the use of many C++ constructs and reduces the likelihood of simple syntactic errors, such as unintended assignments within conditionals. The new binding includes a completely redesigned declaration/definition syntax for types, functions and objects, a simplified template syntax, and changes to several operators and control structures. The resulting syntax is LALR(1) parsable (with no conflicts and no need for semantic feedback to the lexer) and provides better consistency in the specification of similar constructs, better syntactic differentiation of dissimilar constructs, and greater overall readability of code.

1. Introduction

It is widely accepted that the syntax of C/C++, having been evolved over several decades by a large number of contributors, leaves much to be desired [1,2,3]. For example, the declaration syntax which C++ inherits from C (and extends) is so complicated that it is doubtful whether, without the assistance of a manual or source code example, one in ten C++ programmers could correctly declare a prototype for fundamental C++ allocation control function: **`set_new_handler`**[1]. We invite the reader who is familiar with C++ to attempt this exercise before continuing. The answer is given in Appendix A.

In *The Design and Evolution of* C++, Stroustrup observes that "within C++ there is a much smaller and cleaner language struggling to get out" [4] and foresees the development of "other interfaces" to C++. He cites Murray [5] and Koenig [6] who each demonstrated non-textual representations for C++ programs. Whilst such graphical representations of C++ offer considerable assistance in visualization and abstraction, they sacrifice some measure of the convenience, accessibility and portability of a purely ASCII representation.

We propose an alternative text-based syntactic binding (called SPECS[2]) for the existing semantics of the C++ language. The SPECS syntax departs substantially from the existing C++ syntax, particularly in the areas of declarations, definitions, templates, operators, and operator overloading.

Section 2 of this paper discusses some of the language design principles we employed in creating SPECS, whilst Section 3 outlines the SPECS syntax and how it differs from the existing C++ syntax. Section 4 summarizes the remaining similarities between SPECS and C++. Section 5 explains the implementation of the SPECS to C++ translator, while Section 6 discusses its testing and evaluation. Section 7 outlines some future directions. A small but illustrative example of SPECS code is given in Appendix C, and its C++ translation in Appendix D. A User Guide for the translator is given in Appendix E, and a summary of the SPECS grammar is contained in Appendix F.

---

[1] which takes a single argument (a pointer to a function taking no arguments and returning **void**) and returns a similar function pointer (that is: it returns another pointer to a function taking no arguments and returning **void**).

[2] "Significantly Prettier and Easier C++ Syntax"

2. SPECS Syntactic Design Principles

In designing the SPECS syntax we have been guided by four design principles: consistency, differentiation, readability, and formal grammatical simplicity.

The specification of types in C++ provides an excellent example of the lack of *syntactic consistency* that has resulted from the evolutionary development of the language. The following statements all define new types in C++:

```
class RXBase { virtual bool match(string) = 0; };
typedef struct { int x, y, z; } Vector;
typedef int (*NumSrc)();
enum Result { reject, accept, defer };
union Tokens { int num; char* str; };
```

In designing the SPECS binding for C++ we have endeavored to introduce better consistency into all syntactic forms, with particular attention to declaration syntax. In SPECS the above constructs are clearly indicated as creating new types with easily located names:

```
type RXBase : class { func match : abstract (string->bool); }
type Vector : class { [public] obj x, y, z: int; }
type NumSrc : ^(void->int);
type Result : enum { reject, accept, defer }
type Tokens : union { obj num : int; obj str : ^char; }
```

Interestingly, C++ also exhibits the opposite syntactic shortcoming – insufficient *syntactic differentiation* of dissimilar constructs. This is particularly evident in the declaration of functions and objects, where slight variations in component order may completely alter the meaning of a statement:

```
const Vector& (*vectorA)(int,Vector[]);
const Vector* (  vectorB)(int,Vector[]);
const Vector* (&vectorC)(int,Vector[]);
const Vector* &(vectorD)(int,Vector[]);
```

In SPECS the differences between these constructs are clearly indicated and their names are (once again) more easily ascertained:

```
obj  vectorA : ^((int, [] Vector) -> & const Vector);
func vectorB :   ((int, [] Vector) -> ^ const Vector);
obj  vectorC : &((int, [] Vector) -> ^ const Vector);
func vectorD :   ((int, [] Vector) -> & ^ const Vector);
```

Both these design goals help support a third – that of maximizing the overall *readability* of the language. Other notable contributions towards this goal have been:

- reordering declarations to emphasize important information and group together related information (Section 3.1)
- ensuring that, wherever it is important, it can be determined whether a name is a type identifier or not, without reference to prior declarations
- the modification of the template declaration syntax to reduce excess verbosity and better localize relevant information (Section 3.3)
- requiring that all iteration or selection statements are followed by brace-enclosed compound statements rather than single unbracketed statements (Section 3.5)
- the introduction of the keyword `common` to specify "static" class members, thereby reducing the semantic overloading of `static` (Section 3.1.3)

- the replacement of the old-style cast syntax with one similar to the new-style casts (Section 3.4.2)
- the introduction of the **`abstract`** keyword to replace the "`=0`" syntax for pure virtual functions (Section 3.1.4)
- the rebinding of the assignment operator to "`:=`" to better differentiate it from the equality test, which becomes "`=`" (Section 3.4.1)
- the rebinding of the "address of" operator to "`@`" so as to reduce the overloading of "**`&`**" (Section 3.4)
- the introduction of the **`defined_cast`** keyword to make operator conversion declarations more obvious (Section 3.2.2.2)
- the addition of the keywords **`inherits`** (Section 3.2) and **`initially`** (Section 3.2.2.1).

A final design principle for SPECS was *grammatical simplicity*. The language was designed to ensure that the new syntax was LALR(1) parsable, with no shift/reduce or reduce/reduce ambiguities and no requirement for semantic feedback from parser to tokenizer. This constraint considerably simplifies the construction of a portable compiler for the language by allowing us to use the widely available parser construction tools *yacc* and *lex*. The biggest step towards this goal was achieved by ensuring that a type identifier can be identified, wherever this is important, without reference to earlier type declarations. As mentioned above, this has also improved the readability of the language.

3. The SPECS syntax

Sections 3.1 to 3.5 summarize the principal differences between SPECS and (proto-)standard C++. Also see Appendix B for a list of the SPECS keywords. Note that the bracketed symbolic names after each subheading indicate the corresponding section of the working paper for the draft ISO/ANSI C++ standard [7].

3.1. Declarations [dcl.dcl]

The area of greatest difference between SPECS and C++ is declaration syntax. The SPECS binding adopts a number of conventions for consistency across declaration types. Most significantly, all declarations begin with a keyword which identifies the declaration type. This keyword is then followed by the name of the entity being declared, where this is appropriate.

Additionally, all declarations either end in a semicolon or a curly-brace enclosed block. This is also the case for C++ function definitions, however C++ type definitions ending in a curly-brace enclosed block require an additional trailing semicolon. SPECS is consistent across the entire syntax in neither requiring nor allowing such a trailing semicolon.

There are three major types of declarations in SPECS – *type*, *object* and *function*, and one minor type – *lang*. These are described in more detail below.

3.1.1. Type IDs [dcl.name]

Specifying a type in C++ is not simplified by the fact that some of the components of a declaration (such as the pointer specifier) are prefix operators while others (such as the array specifier) are postfix. These declaration operators are also of varying precedence, necessitating careful bracketing to achieve the desired declaration. Furthermore, if the type ID is to apply to an identifier, this identifier ends up at somewhere between these operators, and is therefore obscured in even moderately complicated examples (see Appendix A for instance). The result is that the clarity of such declarations is greatly diminished.

Within SPECS, this problem is overcome by entirely redesigning the type ID mechanism in a manner similar to (but simpler than) that proposed by Anderson [1]. All declaration operators are prefix, right-associative, and are at the same precedence level. Any attached identifiers are separated from the type ID to make them visible. The intention is that the meaning of a type ID can be determined by simply reading it left-to-right, rather than by subtle parsing tricks better left to a compiler than a programmer.

The following are simple C++ abstract declarators:

```
int                  //  integer
int *                //  pointer to integer
int *[3]             //  array of 3 pointers to integer
int (*)[3]           //  pointer to array of 3 integers
int *()              //  function having no parameters and
                     //  returning pointer to integer
int (*)(double)      //  pointer to function of double
                     //  returning an integer
```

The SPECS equivalents are:

```
int                  //  integer
^ int                //  pointer to integer
[3] ^ int            //  array of 3 pointers to integer
^ [3] int            //  pointer to array of 3 integers
(void -> ^int)       //  function having no parameters and
                     //  returning pointer to integer
^ (double -> int)    //  pointer to function of double
                     //  returning an integer
```

The following table describes the operators and specifiers that can compose the declaration:

`^ typeID`	pointer to <*typeID*>
`nestedName::^ typeID`	pointer to <*typeID*> member of <*nestedName*>
`& typeID`	reference to <*typeID*>
`[] typeID`	array of <*typeID*>s with unspecified number of elements
`[constExpr] typeID`	array of <*typeID*>s with <*constExpr*> elements
`(typeID)`	<*typeID*>
`(paramList -> typeID)`	function of <*paramList*> returning <*typeID*>
`const typeID`	constant <*typeID*>
`volatile typeID`	volatile <*typeID*>

Fundamental types are specified slightly differently to C++. A fundamental type name must contain a base type in addition to any size specifier or signed specifier. Thus the C++ fundamental type:

```
unsigned long
```

must be referred to in SPECS as:

```
unsigned long int
```

to ensure that there is no perceived ambiguity as to the base type to which the specifiers refer.

3.1.2. Type declarations and definitions [dcl.type]

As indicated in Section 2, there are numerous kinds of type declarations in C++, each with its own syntax. SPECS is an attempt to unify all of these under one consistent syntax framework. All type declarations begin with the keyword **`type`**, followed by the type name. There are four different kinds of type declaration allowed in SPECS: *simple*, *enum*, *class*, and *union*.

3.1.2.1. Simple type declaration

A simple type declaration is equivalent to a C++ typedef. It associates an identifier with a type ID, as described above in (3.1). It has the syntax:

```
type identifier : typeID;
```

The following is a typical C++ typedef declaration:

```
typedef  int*  IntPtr;
```

The equivalent SPECS declarations is:

```
type  IntPtr  :  ^  int;
```

3.1.2.2. Enum type declaration

An enum declaration is similar to its C++ equivalent, but in a format consistent with the other type declarations. It has the syntax:

```
type optional_enumName  :  enum  {  enumContents  }
```

The following is a simple C++ enum declaration:

```
enum  Colour  {  red=1,  green,  blue  };
```

The equivalent SPECS declaration is:

```
type  Colour  :  enum  {  red:=1,  green,  blue  }
```

Note that there is no trailing semicolon in the SPECS syntax.

3.1.2.3. Class type declaration

A class declaration in SPECS is semantically equivalent to a C++ class. It takes the place of both the C++ class and struct declarations, since the semantics of a struct can be achieved by the addition of a public access specifier at the start of a class type declaration. A class declaration or definition begins with:

```
type className  :  class
```

See Section 3.2 for more details.

3.1.2.4. Union type declaration

A union declaration is semantically equivalent to a C++ union. Its declaration begins with:

```
type optional_unionName  :  union
```

and its syntax is otherwise identical to a class declaration (where this is semantically meaningful). If the optional union name is omitted, an anonymous union is created.

3.1.3. Object declarations and definitions [dcl.meaning]

An object declaration creates one or more variables. These variables can be of any type and need not just be instances of classes. All object declarations begin with the keyword "**`obj`**". The general syntax is:

```
obj objName1, objName2, ... objNameN : type
```

where the type after the colon is one of the four types described above under *Type declarations and definitions* (2.1.2).

The following are C++ declarations:

```
double *x, *y;
enum { green, gold } aColour;
```

The equivalent SPECS declarations are:

```
obj x, y : ^double;
obj aColour : enum { green, gold }
```

Objects can be initialized at construction time. Here are some examples:

```
obj val1, (val2 := 5) : int;
obj myInst(init1, init2) : MyClass::InnerType;
obj array := {1,2,3,4} : [4] int;
obj (ptr1 := 0), ptr2(0) : ^void;
```

In the first example, assignment syntax initialization is used. The brackets around the initialization are optional and are added only to improve visual clarity. The second example uses constructor syntax initialization. The third example uses aggregate assignment initialization. Different initialization syntaxes can be combined within one declaration statement, as in the fourth example. Note however that different types of variables cannot be instantiated in the same declaration. The infamous C++ example:

```
char* c1, c2, c3();
```

has no direct equivalent in SPECS. We consider this to be a feature.

Specifiers can be applied to objects at declaration time. These are listed directly after the colon in the declaration. Non-class-member objects can be declared **`auto`**, **`register`**, **`static`** and/or **`extern`**. Class members can be declared **`common`**, **`mutable`** and/or **`bits(`*`constExpr`*`)`**:

```
obj localMax(1.0) : static register const double;
type MyClass : class { obj ourNextIndex : common int; }
```

Note that the **const** is not a specifier, but part of the type, and must therefore be placed after the specifier sequence.

SPECS reduces the overloading of the **`static`** keyword by introducing a new keyword, **`common`** as a specifier for "static" class members. The other usages of **`static`** as a non-member storage specifier, as in C, are retained.

The **`bits`** specifier is the SPECS representation of the bitfield syntax of C++. The number of bits is placed in brackets after the **`bits`** keyword. The following is a C++ declaration containing a bitfield:

```
class Example { int field:4; };
```

The equivalent SPECS declaration is:

```
type Example : class { obj field : bits(4) int; }
```

An unnamed bitfield can be created by omitting an object name.

3.1.4. Function declarations and definitions [dcl.fct]

A function declaration in C++ has the general structure:

```
opt_specifiers opt_returnType functionName ( opt_paramList );
```

as in the examples:

```
extern char* getString(int);
static computeNextValue();
```

We identify three main problems with the syntax of C++ function declarations:

- Function declarations are very similar in structure to variable declarations. In cases where the type of a variable involves a pointer to a function, the distinction can become very subtle, often involving only slight differences in bracketing.

- The name of a function has no uniform location within the declaration, and in more difficult examples can be embedded within levels of bracketing. Looking for a function within a mass of code can be a challenge.
- The return type is placed before and some distance away from the parameter list, or may be only implied. This makes it difficult to quickly ascertain the complete type of a function.

The SPECS syntax for function declaration tackles each of these problems. All function declarations begin with the keyword "`func`", followed by the name of the function. The general syntax is then:

```
func functionName : opt_specifiers ( paramList -> returnType );
```

The *paramList* is an optionally-bracketed, comma-separated list of parameters, in one of the forms:

```
paramName : paramType
paramName := initVal : paramType
(paramName := initVal) : paramType
```

Variable length parameter lists can be declared using the ellipsis notation ("`...`"), as in C++. If a function takes no parameters, the `void` keyword is used in place of the parameter list to indicate this. The following are C++ function declarations:

```
void fn1();
char* fn2(char* param1, int param2);
const T& fn3(int (*param1)[10]);
```

The equivalent SPECS declarations are:

```
func fn1 : (void -> void);
func fn2 : ((param1 : ^char, param2 : int) -> ^char);
func fn3 : ((param1 : ^ [10] int) -> & const T);
```

Specifiers can be applied to functions at declaration time. These are listed in sequence directly after the colon in the declaration. The specifier for a non-class-member object may be either `static` or `extern`. Class-member specifiers are: `inline`, `virtual`, `abstract`, `explicit` (constructors only) and/or `common`. The `common` keyword replaces the `static` keyword in the same way as with object declarations (section 3.1.3).

The `abstract` keyword specifies that a member function is pure virtual. It is mutually exclusive with `virtual`, and can only be applied to declarations/definitions within a class:

```
type MyClass : class
{
     func printMe : abstract (& ostream -> void);
}
```

The equivalent C++ is:

```
class MyClass
{
     virtual void printMe(ostream&) = 0;
}
```

3.1.5. Language declarations [dcl.asm, dcl.link]

A language declaration in a SPECS program specifies a (link to a) non-SPECS code fragment. It has the general structure:

```
lang "languageName" { declarationSeq }
```

where the set of allowed language names is implementation dependent, but contains at least **asm**, **C**, **C++** and **SPECS**. The **lang** keyword provides a unification of the **asm** (assembly directive) and **extern** (external linkage) declarations of C++, whilst eliminating the semantic overloading of the latter.

Within an `asm` language declaration, assembly directives are either newline or semicolon terminated (or both):

```
lang "asm"
{
    mov eax, ebx; xchg ecx, edx
    shl eax, 7;     // This semi-colon not required
}
```

The equivalent C++ would be:

```
asm("mov eax, ebx"); asm("xchg ecx, edx");
asm("shl eax, 7");
```

The behavior of the **C** language declaration is identical to the corresponding **extern** (external linkage) declaration in C++. By extension, standard C++ syntax code is expected within a **C++** language declaration, which is useful when writing SPECS programs that include one or more C++ standard libraries:

```
lang "C++"
{
#include <iostream.h>
#include <fstream.h>
}
```

3.1.6. The using declaration [namespace.udecl]

In C++, the **using** declaration has the form:

```
using name;
```

where *name* is some (optionally scoped) specified name. SPECS provides three different **using** declarations, namely:

```
using type name;
using obj  name;
using func name;
```

These allow better error checking to be performed and give a greater indication to the reader of what kind of name is being introduced into the scope.

3.2. Class declarations and definitions [class]

A class declaration in SPECS is semantically equivalent to a C++ class. It takes the place of both the class and struct declarations, since the semantics of a struct can be achieved by the addition of a public access specifier at the start of a class type declaration. A class declaration has one of the following formats:

```
type className : class;

type className : class opt_inheritanceList { /* ... */ }
```

where the first introduces the class name, and the second defines the class. The inheritance list is optional in the second case, and specifies a list of classes from which this class inherits properties. It has the form:

```
inherits baseClass, baseClass,  ...etc
```

where the base class specifications may include one of the access specifiers: **public**, **protected** and **private**, and/or the inheritance specifier **virtual**, with the same semantics as in C++.

The following SPECS class declaration:

```
type ListClass : class
    inherits public ListBase, virtual ListImpl
{ /* ... */ }
```

is equivalent to the C++ declaration:

```
class ListClass : public ListBase, private virtual ListImpl
{ /* ... */ };
```

3.2.1. Member access control [class.access]

A member of a class can be declared **`public`**, **`protected`** or **`private`** (the default). Within a class, only one of these access specifiers will be the active one at any time. To change the current specifier within a class declaration, a directive of the form **`[`*`accessSpecifier`*`]`** is used:

```
type TestClass : class
{
     // members declared here are private
[public]
     // members declared here are public
}
```

3.2.1.1. Friends [class.friend]

In C++, a friend declaration is performed by placing the keyword **`friend`** in front of the declaration. We believe that this does not sufficiently differentiate those declarations that refer to members and those that refer to friends. To this end, the usage of the **`friend`** keyword has been altered. In SPECS, **`friend`** is used in exactly the same way as the access specifiers described in (3.2.1). As such, the directive **`[friend]`** sets all following declarations to be friend declarations, until the end of the class or the next access specifier is reached. An example is:

```
type testClass : class
{
     // members declared here are private
[friend]
     // functions and types declared here are friends
}
```

Note that many types of declarations are not permitted to be declared as friends, and declaring them within a friend declaration block will cause the program to be ill-formed and generate a diagnostic.

3.2.2. Special member functions [special]

3.2.2.1. Constructor and destructor declarations

In C++ the names of a constructor and destructor of a class are respectively the name of the class, and a tilde followed by the name of the class. Because these names will differ from class to class, it can be difficult to identify these members at a first glance. SPECS renames them to **`ctor`** and **`dtor`** respectively, to rectify this problem.

A constructor definition has the general formats:

```
func ctor : optional_specifiers ( paramList )
     optional_initializer_list
     { /* ... */ }
```

Note that, as in C++, a constructor has no specified return type.

An initializer list is introduced with the keyword **initially**:

```
func ctor : (size : int, name : ^char)
     initially BaseClass::ctor(), mySize(size), myName(name)
     { /* ... */ }
```

A destructor definition has the general format:

```
func dtor : optional_specifiers ( void )
     { /* ... */ }
```

It is illegal to declare a constructor or a destructor outside of a class or named union. However a definition of a class's constructor or destructor that is outside the class body is legal. For example:

```
func MyClass::dtor : (void)
     { /* ... */ }
```

3.2.2.2. Operators and casts

The C++ operator overloading semantics are unchanged in SPECS but the syntax is modified to conform to the function declaration syntax described above and the renamed operators listed in Section 3.4.1 below. In addition, two keywords, **`pre`** and **`post`**, have been introduced to simplify the overloading of the pre- and post- increment and decrement operators, replacing the awkward "dummy integer" syntax:

```
type MyInt : class
{
     obj myVal : int;
[public]
     func operator= : (i:& const MyInt -> & MyInt)
         { myVal = i.MyVal; return this; }
     func operator pre ++ : (void -> & MyInt)
         { myVal++; return this; }
     func operator post ++ : (void -> MyInt)
         { obj oldVal(this) : MyInt; myVal++; return oldVal; }
}
```

Note that in SPECS (as illustrated in the above example) the identifier **`this`** within a member function acts as a reference, not as a constant pointer (as in C++).

In C++, a declaration of an operator conversion-to-*typeID* function has the name **`operator`** ***`typeID`*** and no return type. The SPECS declaration of an operator conversion function has the name **`defined_cast`**, and the target conversion type as the return type. The following is a typical C++ operator conversion function declaration:

```
class MyClass
{
public:
     operator int();
}
```

The equivalent SPECS declaration is:

```
type MyInt : class
{
[public]
     func defined_cast : (void -> int);
}
```

We believe this to be more straightforward than the C++ approach, and more consistent with new-style cast naming practices.

3.3. Template declarations and definitions [temp]

With ever increasing use of template libraries, templates are becoming a central feature of C++. However the syntax of C++ template templates is not always easy to comprehend, especially when nested templates are used:

```
template<class T> class string
{
public:
     template<class T2> int compare(const T2&);
};
```

It is even more cumbersome to define such a member of a template outside the template body :

```
template<class T> template<class T2>
int string<T>::compare(const T2&) { /* ... */ }
```

The problem stems from the fact that a template declaration has its template parameter list separated from the name that is being parameterized. SPECS alters this syntax so as to eliminate this separation and the excess verbosity of such declarations.

The first change is from simple angle brackets ("**<**" and "**>**") to composite brackets ("**<[**" and "**]>**"). Replacing single character brackets with two character brackets is not a change that was taken lightly, but it does solve three C++ problems and at the same time clearly delineates the parameterization of a template. The problems solved are:

- There is no longer a need to bracket template arguments containing the "**>**" operator, as in the C++ declaration **`TemplateClass<(1>2)>`**, since the SPECS version: **`TemplateClass<[1>2]>`** is unambiguous.
- It is no longer necessary to put a space between successive closing template brackets, (for example: **`array<auto_ptr<X> >`** in C++) . In SPECS, **`array<[auto_ptr<[X]>]>`** is unambiguous.
- The **`template`** keyword is not needed to disambiguate a member template function call, as in C++:
  ```
  x = p->template alloc<200>();
  ```
  The SPECS equivalent is:
  ```
  x := p^.alloc<[200]>();
  ```

The biggest change to the template declaration syntax is the movement of the template parameter list from the start of the declaration to just after the name being parameterized. The parameters within the list must be of one of the following type forms:

```
type identifier
type identifier := default_typeID
type identifier <[ templateParamList ]>
type identifier <[ templateParamList ]> := default_templateName
```

or of the following non-type forms:

```
obj identifier : typeID
obj identifier := default_value : typeID
```

3.3.1. Class templates

The general form of a template class is:

```
type className <[ templateParamList ]> : class opt_derivedList
{
    // template members...
}
```

The following C++ template examples:

```
template<class K, class V> class Map { };
template<class T1> template<class T2> class Outer<T1>::Inner { };
```

are written in SPECS as:

```
type Map<[type K, type V]> : class { }
type Outer<[type T1]>::Inner<[type T2]> : class { }
```

Specializations of these templates in C++ would look like:

```
class Map<int,char*> { };
template<class T1> class Outer<T1>::Inner<int> { };
```

The SPECS versions are easier to identify as type declarations:

```
type Map<[int, ^char]> : class { }
type Outer<[type T1]>::inner<[int]> : class { }
```

3.3.2. Function templates [temp.fct]

The general form of a function template is:

```
    func functionName <[ templateParamList ]> :
            opt_specifiers ( paramList -> returnType );
```

Usage is analogous to class templates (3.3.1).

3.4. Expressions [expr]

Changes to the C++ expression syntax fall into two categories – operator changes and changes to the casting syntax.

3.4.1. Operator changes

The C++ equality test (`val1 == val2`) has been changed in SPECS to `val1 = val2`, as this binding for "`=`" is more consistent with mathematical usage. The inequality operators ("`!=`", "`<`", "`<=`", "`>`", "`>=`") are unchanged.

As a consequence of the change in the equality operator, the C++ assignment operator (`ref = val`) becomes `ref := val` in SPECS[1]. As might be anticipated, the compound assignment operators become "`+:=`", "`-:=`", "`*:=`", etc.

The C++ unary prefix address-of operator (`&rval`) becomes a postfix operator in SPECS: `rval@`. The unary prefix pointer dereference operator (`*ptr` in C++) also becomes a postfix operator: `ptr^`. As a consequence of this latter change to postfix notation, the C++ binary dereference-and-select-member operator (`ptr->member`) is no longer required in SPECS as the syntax `ptr^.member` suffices. Likewise member selection through member pointers (`ptr->*memptr` and `ref.*memprt` in C++) become `ptr^.(memptr^)` and `ref.(memprt^)` respectively.

As a consequence of the use of "`^`" as the pointer dereference operator, the C++ binary bitwise exclusive-or operator (`bits1 ^ bits2`) has been changed to `bits1 ! bits2` in SPECS. The rationale for the choice of "`!`" is the analogy to "`!=`", the logical equivalent of xor.

The use of operators `new` and `delete` is unchanged in SPECS, except in the case of the placement syntax. A new keyword, `placement`, is reserved and the following C++ code:

```
    ptr = new (myLocation) int[10];          // Placement new
    ::operator delete[](ptr,myLocation);     // Placement delete
```

becomes, in SPECS:

```
    ptr := new [10] int placement(myLocation);
    delete [] ptr placement(myLocation);
```

This both improves the readability of the expression and considerably simplifies the grammar. Note too that the array type on which `new` operates must conform to the declaration syntax described in Section 3.1.1, and so the array size precedes the element type. This has the useful side-effect of improving the consistency of the location of square brackets in calls to `operator new[]` and `operator delete[]`, which, unlike C++, immediately follow the operator name in all cases in SPECS.

3.4.2. Changes to the casting syntax

New-style casts in C++ provide a clear indication that a cast is in progress, and of the purpose of that cast. However the ISO/ANSI C++ standard committee has not deprecated the use of old-style casts. Hence, SPECS allows old-style casts, but requires a clearer and more consistent syntax for their use:

```
    cast<[typeID]>(Expression)
```

[1] Note that "`<-`" was originally our preferred candidate for the assignment operator, but this binding creates significant problems because assignments can occur in logical tests, leading to ambiguities such as:

```
    if (i<-10) { cout << "big negative" << endl; }     // Always printed?
```

Note that this modified syntax for old-style casts does nothing to improve their safety. Where supported, new-style casts should be used in preference. Their syntax is also slightly modified, in line with the change to compound template brackets in SPECS:

```
static_cast<[typeID]>(Expression)
dynamic_cast<[typeID]>(Expression)
const_cast<[typeID]>(Expression)
reinterpret_cast<[typeID]>(Expression)
```

3.5. Statements [stmt.stmt]

The **`while`**, **`do-while`** and **`for`** loops retain their C++ syntax in SPECS, except that a single unbracketed statement is no longer permitted as a loop body (that is, the body of a loop must be a block in SPECS). Likewise, in the **`if`** statement, the code to be executed if the condition is true must be a brace-enclosed block of zero or more statements, and the statement following an **`else`** keyword must either be a (cascaded) **`if`** statement or a brace-enclosed block.

The **`switch`** statement in SPECS has been considerably altered from its C++ equivalent. It consists of zero or more specified cases and an optional default case, For example:

```
 switch ( nextValue )
{
     case 1:
     { cout << "Unity" <, endl; }
     case 2,4,6,8:
     { cout << "Even digit" << endl; }
     case 3,5,7,9:
     { cout << "Odd digit" << endl; }
     default:
     { cout << "Too big" << endl; }
 }
```

Each case consists of a list of one or more (integral) constant expressions, followed by a brace-enclosed block. If the condition matches any of the constant expressions belonging to a case, then the corresponding block of code following the colon will be executed. After the block executes control jumps to the end of the switch statement (*not* to the next case, as in C++). The **`break`** statement will also cause control to jump to the end of the switch statement. The **`continue`** statement can be used in a case block and causes control to jump immediately to the beginning of the next case block, thereby implementing a more general, but safer form of fall-through than the C++ default behaviour.

4. Commonalities with C++

By this stage the reader may feel that SPECS has little in common with its parent C++. In fact, the two languages are semantically isomorphic and significant portions of the two languages are syntactically identical. Features common to C++ and SPECS include:

- All inbuilt types are identically named and implemented. All literal values are specified in exactly the same way.
- The same set of operators (with the same precedences) are available for overloading. All binary operators maintain the same associativity in both languages.
- All scoping rules, temporary lifetimes, overloading constraints, function call resolution mechanisms, implicit conversions, access defaults and restrictions, and control structure semantics (except switch statement fall-through) are identical.
- The same preprocessor (cpp) and standard libraries are used for both languages.
- RTTI is identically implemented in both languages, and uses the same syntax.
- The exception handling mechanism and keywords are the same. The specification syntax is also identical except where syntactic differences in type or function specification preclude this.
- Namespaces have identical semantics and syntax in both languages.

5. Implementation

In order to test the usability of SPECS it was essential to develop a software translator that would convert SPECS code into C++ code. This would:

- Provide a testbed for the development and design of the SPECS syntax
- Provide evidence of the feasibility of the SPECS syntax design
- Allow programs to be written in SPECS and be compiled into executable form via C++ intermediate code

We have been successful in this aim and have produced a complete compilation system that compiles SPECS into native executable format, with the aid of the system C++ preprocessor and compiler. The translator has been thoroughly tested and is considered stable. It fully implements all draft-standard C++ semantics as defined in [7]. There are subtle changes to some constructs, as discussed in Section 3, but these are almost always to correct the error-prone semantics of the relevant construct.

5.1 Translator Environment

The SPECS translator system consists of the SPECS translator, the C++ preprocessor, the C++ compiler, and a SPECS loader program which controls the execution of the other components

5.1.1. Loader Program

The loader program is the program that is executed by the user. The translator itself is never directly executed by the user in normal practice. The loader program controls the translation process by calling the other units (i.e. preprocessor, SPECS translator, C++ compiler) as required. The loader program has a few main tasks to perform:

- Interpreting of command line arguments and determining whether they refer to the preprocessor, the SPECS translator, the C++ compiler, or the loader itself.
- Managing the list of files that are to be processed and tracking the initial types of the files, the desired output file type (i.e. executable, compiled object code, preprocessed), and the current state of each file.
- Starting child processes containing the preprocessor, the SPECS translator, or the C++ compiler, as required, and passing them their relevant command line arguments.
- Catching return error codes in any of the child processes and reporting success, failure due to errors, or abnormal termination.
- Handling and creating temporary files for each stage, and deleting the appropriate files upon either success or failure.

The loader program is designed to work with the GNU **cpp** preprocessor and **g++** compiler. It recognizes all command-line arguments of these programs (including multiple argument flags) and directs them to these programs when they are executed. It could be ported to another preprocessor and/or compiler with a moderate amount of effort.

The translation process need not translate from SPECS source all the way to an executable. A set of command line arguments to the loader program will tell it to stop at particular stages. These stages (and their default extensions) are:

- Preprocessed and not translated to C++ (.cpp)
- Preprocessed and translated to C++ (.ii)
- Compiled but not assembled (.s)
- Compiled and assembled (.o)
- Compiled and linked (no extension)

The loader creates child processes by the standard UNIX technique of executing a fork followed by an execv, and then doing a wait until the created child process exits, at which time it reads the returned error code.

5.1.2. Interface to Preprocessor

The SPECS source file is first preprocessed to strip comments and expand include directives. Output from preprocessors does not follow any standard format with regard to embedded line and pragma directives. Since the SPECS translator must read and process these directives, it must be set up for the particular output format in use. The current implementation is designed to interface with GNU **cpp** preprocessor, which is a component of the publicly available free GNU C and C++ compilers. Substitution of a different preprocessor may require that one simple routine in the lexer be changed to enable it to read the new preprocessor output format.

5.1.3. Interface to C++ Compiler

The SPECS translator outputs C++ code that does not require a second preprocessing pass. As such, the code is likely to contain embedded line and pragma directives. These may be from the preprocessing stage or may be added by the SPECS translator (for example, to ensure accurate line numbers in later error messages). As different C++ compilers encode these directives differently in preprocessed code, the SPECS translator must output these directives in the correct format. The current implementation is (once again) designed to interface with GNU **g++** compiler. Use of a different compiler may require that one simple routine in the output interface be changed to enable it to write the new format.

5.2. Translator Internal Overview

The components of the SPECS translator itself are the lexical analyzer, the parser, the code generation system, the error detection and handling system, and the output interface.

5.2.1. Parser and Lexical Analyzer

The lexer (lexical analyzer) and parser are tightly connected components and together make up the first logical step in the translation process. In the current implementation, the lexer is implemented in GNU *flex* (an improved clone of *lex*), while the parser is implemented in GNU *bison* (an improved clone of *yacc*). The standard UNIX utilities *lex* and *yacc* could be used instead with some trivial modifications.

The lexer scans through its input files and breaks them down into a stream of tokens which each represent some pattern of text. The parser controls the lexer by only requesting tokens each time it is ready for a new one, at which time the lexer runs until it has found the next token. There is no semantic feedback required from the parser back to the lexer, since the grammar is context-free and unambiguous. More specifically it is LALR(1) with no shift/reduce or reduce/reduce conflicts.

The grammar of SPECS is summarized in Appendix F. The interested reader will find it useful to read sections 5.2.1.2 and possibly 5.2.1.1 before delving into the grammar.

5.2.2. Development of Grammar and Parser

The grammar of SPECS is a large and complicated one being somewhat longer that the draft ANSI standard C++ grammar. The standard C++ grammar has several deficiencies:

- It allows a considerably larger set of constructs than are supported by the C++ language. Thus a great deal of work must be done in semantic analysis of C++ just to catch what are simply syntax errors.
- The C++ grammar is ambiguous. The C++ grammar has a number of cases that require the application of 'disambiguation rules'. There are case where large amounts of lookahead are required to correctly parse a token. An LALR(1) implementation of C++ without significant semantic feedback is not possible. Finally, it arbitrarily requires additional keywords in some constructs to allow correct parsing.
- The C++ grammar makes no attempt at differentiating different types of constructs within the grammar.

The SPECS grammar is a much tighter superset of the SPECS language than the C++ grammar is of the C++ language. The SPECS grammar is entirely unambiguous and is LALR(1) parsable with no semantic feedback. Within the SPECS grammar, the philosophy of consistency among similar constructs and differentiation of dissimilar constructs has been adhered to. This lead to a larger grammar, but also, we believe, to a more clearly-defined description of this aspect of the language.

The task of creating the SPECS grammar with its 500 or so rules was a long process of refinement and testing and was intimately tied to the development of the parser. Though its grammar bears some similarities to the C++ syntax, it is radically different in all the areas where SPECS differs from C++, and is often even different in those areas where the two syntaxes are identical. The latter differences are due to the process of disambiguating the grammar to make it LALR(1).

The sequence of steps that contributed to each iteration of the refinement of the grammar were:

- Determine the full semantics of a C++ construct or group of constructs. This entails understanding the obscure and undesirable uses of the construct, as well as the mainstream usage.
- Design a construct for SPECS that provides the semantics of the C++ construct.
- Add this construct to the grammar and disambiguate the grammar. Sometimes adding the new rules will cause ambiguities in the grammar to arise. The grammar must then be modified by either changing the construct or changing other constructs so that the new construct does not cause ambiguities. This is a complicated and time-consuming process.
- Add tests of this construct to the test suite and thoroughly test the parsing of it. At this stage subtle problems can be detected that must be rectified.
- Ensure that the SPECS construct contains enough information such that it is possible to generate the desired C++ construct during code generation.

5.2.3. Other Issues and Limitations

The main limitations of the grammar derive from the constraint that the translator cannot know the names of all types that have been declared. A full compiler would track all type declarations (and other declarations), and could then return from the lexer different tokens for non-type identifiers and type identifiers, since it has seen the type identifiers before. The translator does not store this information, and thus cannot make use of it. Doing otherwise would require parsing all C++ code that is included through a **`lang`** declaration, since types could be declared within this code.

This creates a problem. There are three points in the grammar where either an *expression* or a *TypeID* could be used. These are within a **`typeid`** RTTI call, with in a **`sizeof`** call, and as a template argument. In each case there is an ambiguity with parsing things such as **`scope1::scope2::name`** or even just **`name`** since these symbols could be either referring to a type or a non-type. To solve this problem a new *TypeID* nonterminal called a *RestrictedTypeID* is created. This nonterminal accepts everything a *TypeID* does except for those cases that cause an ambiguity. Since all we wish to do in those contexts is to pass the identifier name to the C++ compiler without modification, it does not matter what it is parsed as, and the parser simple opts for treating it as a non-type identifier.

There is one further problem that cannot be solved with this approach: an additional ambiguity between a function type declaration such as:

```
(scope1::myType, myType2 -> double)
```

and an expression list constructed with the comma operator:

```
(scope1::obj1, obj2) + obj4
```

Because such a list can be of arbitrary length, this ambiguity cannot in general be resolved with any fixed number of lookahead tokens. Consequently, function type declarations must be removed from the *RestrictedTypeID* rules, meaning that function type declarations cannot be used in any of the three locations listed previously. Note howver that this restriction does not apply to the vastly more common cases of pointer-to-function or reference-to-function declarations. The functionality that is lost is very rarely used, and a simple workaround is to do:

```
type dummy : (double -> double);
... myTemplate<[dummy]> ...
```

instead of:

```
... myTemplate<[(double -> double)]> ...
```

An implementation of SPECS that did record declarations of all type names would be able to use this information to remove the entire *RestrictedTypeID* complication. Note that this solution would require complete parsing of any included C++ code.

5.3. Code Generator

The code generation process involves the creation of an output parse tree in addition to the input parse tree that is implicitly created during the parsing process. This output parse tree is a tree representation of the C++ code that will be output. It is built bottom up, in conjunction with the parsing of the input stream.

When each rule is reduced during the parsing of the input stream, an object is returned that represents the C++ parse tree equivalent of that SPECS rule. If this rule then becomes a part of a larger rule, the object that represents the C++ code of that rule is used as part of another object that represents the C++ code of the larger rule. Thus as the input stream parses bottom up, the output parse tree is built bottom up until it represents the C++ of a global level declaration. At that time, the tree stored in the top-level object is recursively traversed to produce a text representation. The effect is that evaluation requests cascade all the way down the tree, and every node is evaluated.

The nodes of the output tree are given the name 'node operators'. They are all objects of classes derived from the class **`Translated`** The **`Translated`** class declares an abstract (pure virtual) member function called **`print`** that all its descendants must define. The **`print`** function is passed the output stream object which it is to print to, and will evaluate and print a text representation of the node to that stream, recursively calling the **`print`** member of any sub-nodes it contains.

This technique will handle translations that are rearrangements or modifications of the input without more complicated dependencies. However, more complicated translation processes are performed by the **`TypeID`** node operator (for type IDs) or by moving things across scope levels using the declaration record stack. Additionally, SPECS's new switch statement semantics require some small code stubs be included for each case. This is done by the **`SwitchCase`** node operator. All of these subclasses of Translated are discussed in more detail below.

5.3.1. Simple Node Operators

There are a large number of simple node operators defined within the SPECS translator. Examples of these are:

- The **`Empty`** operator that evaluates to nothing.
- The **`NewLine`** operator that allows any number of carriage returns to be placed before and/or after its parameter node operator.
- The **`SimpleBlock`** hierarchy of classes which are constructed with a text string, and will print that string when evaluated. For clarity, this is subclassed into a number of more specific derived classes:
    - **`Keyword`**
    - **`Operator`**
    - **`Identifier`**
    - **`IntegerConstant`**
    - **`CharacterConstant`**
    - **`FloatingConstant`**
    - **`StringLiteral`**
    - **`BooleanConstant`**
    - **`AsmDirective`**
- The **`DeclBlock`** operator will print its parameter node operator enclosed in curly brackets like a block of declarations.
- The **`Composition`** operator is the most commonly used. It allows its parameter node operators and text strings to be concatenated.
- The Bracketed hierarchy provides the ability to bracket the parameter node operator. It has derived classes:
    - **`RoundBracketed`**
    - **`SquareBracketed`**

    - **`AngleBracketed`**
    - **`CurlyBracketed`**
- The **`CastBase`** hierarchy provides the ability to produce a new-style cast expression given two node operator parameters – the type to cast to, and the expression to cast. It has derived classes:
    - **`DynamicCast`**
    - **`StaticCast`**
    - **`ReinterpretCast`**
    - **`ConstCast`**

5.3.2. `TypeID` Node Operator

The **`TypeID`** node operator handles the complications of generating C++'s 'inside-out' declaration syntax. It breaks the declarations into three parts - type part, left part, and right part. These are located as follows:

`typePart leftPart` **`name1`** ***`rightPart, leftPart`*** **`name2`** ***`rightPart , ...`***

for as many names as there might be. The **`TypeID`** node operator is derived from **`Translated`** and thus has a **`print`** member function and can be passed as a parameter to any function that takes a node operator. However it provides additional functionality to perform its task.

The **`makeFunc`** member function takes a node operator as a parameter that represents a parameter list. This is concatenated to the left side of the right part. The **`makeArray`** member function performs this same function with an array declarator (i.e. **`[num]`**).

The **`makePointer`** member function is more complicated in that must add bracketing if the last call was a **`makeFunc`** or **`makeArray`**. It concatenates the pointer declarator it receives to the right side of the left part.

The `TypeID` node operator also has the facility to hold a list of names so that they can be output correctly, each surrounded by a left part and a right part, as above.

5.3.3. `SwitchCase` Node Operator

The `SwitchCase` node operator is responsible for outputting the code stubs to provide the new semantics of the switch statement (see section 3.5). It is passed a node operator for a block of code, and adds the necessary code stubs to it. This method of changing the semantics of the switch statement causes a slight performance degradation, and a true compiled implementation of SPECS would generate this construct directly.

The following SPECS code:

```
switch  (val)

{
      case 1,2,3 : { cout << a; }
      default :    { cout << b; }
}
```

produces the C++ code:

```
switch  (val)
{
      int  _SPECS_SWITCH_FLAG_;

     case 1: case 2: case 3:
          _SPECS_SWITCH_FLAG_ = 1;
          while(1) { if (_SPECS_SWITCH_FLAG_--) { cout << a; } break; }
          if (!_SPECS_SWITCH_FLAG_) break;

     default:
          _SPECS_SWITCH_FLAG_ = 1;
          while(1) { if (_SPECS_SWITCH_FLAG_--) { cout << b; } break; }
          if (!_SPECS_SWITCH_FLAG_) break;
}
```

This will produce the desired effect, including correct handling of **`break`** and **`continue`** keywords in the case block.

5.3.4. Declaration Record Stack

The declaration record stack is an important tool for keeping track of information during the translation. It provides a host of query functions and data storage functions on the current scope record. Each time a function, obj, or type declaration is entered, a new entry is created on the stack, and when that declaration is left, the stack is popped and the information deleted. The stack provides the ability to walk back through the records of all the enclosing scopes, providing valuable information that is required for many parts of the code generation and error checking. The stack is used heavily as a place to store information to be accessed across a particular declaration.

The declaration record stack is implemented using a somewhat sophisticated linked list class that holds the records. A clean interface is provided to the stack to ensure that the all users of it are well behaved.

5.3.5. Output Interface

The output interface ensures that pretty-printing of the output is performed, and that line directives are correctly output if they are desired. It also contains a number of state flags that are used to control the style of output.

The main functionality it provides are the stream manipulator **`NEXTL`**, which is used in place of **`endl`**, and the functions **`incIndent`** and **`decIndent`**

5.4. Error Checking and Handling

Errors can be detected by the SPECS translator at one of two stages – during the parsing of the code, or during an attempt to generate code. All errors messages are passed to an interface which is responsible for printing the message, the file name, and the line number of the error. Translation does not terminate immediately when errors are found unless the translation attempt is complete or a maximum number of errors is reached.

5.4.1. Errors Found During Parsing

The parser generation tool *yacc* (or in this case, its clone *bison*) contains a facility to add error recovery rules to guide error recovery in case of a parse error. The SPECS parser contains many such rules which are placed to provide the most accurate possible error messages, while attempting to avoid runs of consequential error messages. It is successful at the first, and moderately so at the second. Each parse error will cause a diagnostic message to be printed.

5.4.2. Errors Found During Code Generation

During code generation, checks are made to catch errors that would cause code generation to fail or where a SPECS construct is otherwise incorrectly used. Only certain classes of errors can be detected without deeper semantic analysis. Those that cannot be detected by the translator will likely be found by the C++ compiler in the next stage.

The following is a list of all errors found by the SPECS translator during code generation. Generally these are detected upon reaching a keyword or construct that is not consistent with the current declaration context, and is found by checking the declaration record stack. Others are found when an attempt is made to generate some code but the necessary information is not registered with the current declaration record. The list of errors is:

- **attempt to apply 'common' to a non class member**
- **attempt to apply 'static' to a class member - use 'common'**
- **can only declare a method as abstract**
- **can't apply both 'abstract' and 'virtual' to a method**
- **can't create an object of undefined type**

- **can't declare a template here**
- **can't declare a template this way**
- **can't declare an obj with function type**
- **can't generate an unresolved template**
- **can't give a definition to an abstract method**
- **can't have a ctor in an unnamed class**
- **can't have a dtor in an unnamed class**
- **multiple 'bits' specifiers**
- **not a valid friend declaration**
- **unnamed object must be a bitfield**
- **unscoped ctor outside class/union**
- **unscoped dtor outside class/union**
- **used 'using' in a function when its output is turned off**

These should all be self-explanatory, apart from the last error message. This refers to a case where, for compatibility with older C++ compilers, the translator is in a mode where the `using` keyword is treated as a no-op and is not passed through to the output. If this is done within a class, an access declaration [class.access.dcl] is the result. However an access declaration cannot be contained within a function, and the error message is produced if an attempt is made to do so.

5.4.3. Limitations of Error Checking

There are numerous classes of deeper semantic errors that will not be detected by the SPECS error detection mechanism. However this is of no consequence as these errors are not directly relevant to SPECS and may be left to the C++ compiler to detect. SPECS outputs line directives to the C++ compiler so the correct line number for diagnostics will be produced.

However there is a troublesome class of errors which neither the SPECS translator nor the C++ compiler can detect. They are cases where two or more SPECS constructs map onto the same C++ construct. If the SPECS translator cannot catch the error then the program will compile without er-roreven though it is semantically ill-formed. There are two cases of this in the SPECS translator.

The first case is in regard to the following SPECS declarations:

```
using  type myName;
using  func myName;
using  obj  myName;
```

These all map onto the C++ declaration:

```
using  myName;
```

and thus each of the three will work in any context, even though only one should. Catching this would require a great deal of extra semantic analysis, as described in section 5.3.2.

The second case is where a class constructor should explicitly construct its base class using the syn-tax:

```
initially  baseClass::ctor(val)
```

but a program containing the incorrect initialization:

```
initially  baseClass(val)
```

will produce the correct result, since both versions map to the same C++ syntax.

6. Testing and Evaluation

Testing was carried out regularly with the use of a simple but effective test suite. This suite was kept up to date with the syntax by mirroring any changes in the syntax with changes to the contents of the suite. The suite contains the following types of tests:

- Usage of all constructs, in unusual as well as usual ways
- Examples of C++ code from the draft-standard (converted to SPECS) to test the more complicated areas such as templates and exceptions
- Additional C++ examples from a variety of source (converted to SPECS)

Testing of a tool with much of the functionality of a compiler is a difficult process. An important aspect of testing is the feedback from users of the translator. To date a considerable number of small but real-world examples have been translated, and there have been no bug-reports or problems of any kind.

The performance of the translator is good, comprising only a small fraction of the time of the overall compilation process in all tested cases on a variety of platforms. As a result, the differences in compilation times of a SPECS program and a C++ program is minimal. The memory and system demands of the translator are significantly less than those of a typical C++ compiler.

The SPECS translator thus fully satisfies its defined purpose of being a testbed and proof of concept for the SPECS syntax. In its current state it allows development of systems of arbitrary size and complexity and efficiently and reliably performs its task.

7. Future Work

We have demonstrated the feasibility of the SPECS syntax as well as the possibility of producing a SPECS to C++ translator. A logical extension to this work would be to design a complete new language which applies a SPECS-like syntax to a simpler and cleaner set of semantics than that of C++. The same philosophy of improving consistency and differentiability could apply here. Another goal would be the reinforcing of good design principles by providing facilities for clean interfaces and class frameworks. A portable compiler for this language would be the ideal way to show its feasibility.

A simpler extension to this work would be to provide a SPECS-like syntax interface for Sun Microsystem's Java language. The Java language meets the criteria specified above in that it is syntactically similar to C++ (and therefore amenable to a SPECS-like syntax), but has simpler and cleaner semantics than C++. It has many other advantages over C++, and may be preferable for certain classes of problems.

Conclusion

In implementing a new text binding for the semantics of C++, we have enjoyed the unparalleled advantage of hindsight and the freedom to step beyond the restrictions of the evolutionary path of C++. Most significantly we have rejected the "failed experiment" of the C declaration notation, in favour of a more Algol/Pascal-like approach. We have also taken the opportunity to clean up other error-prone constructs and vestigial unpleasantnesses, such as fallthrough in a switch, the declaration of objects of (subtly) differing types in a single declaration, single statements as control structure bodies, **`this`** as a constant pointer (rather than a reference), the overuse of the "**`&`**" symbol and **`static`** and **`extern`** keywords, the "**`=`**"/"**`==`**" confusion, and the cryptic "**`=0`**" syntax for pure virtual functions.

The experience of syntactically redesigning a language of the complexity of C++ has been challenging and (at times) frustrating, and has left us with nothing but admiration for the designers of any real-world language, who must not only contend with all the issues and choices we have faced, but must also address the much harder task of simultaneously designing a consistent, powerful, and comprehensible semantics. It is our hope that the example of SPECS will encourage such language designers to recognize that language syntax is the *physical* interface between programmer and computer and, as such, demands just as much care and design as the language semantics, which is the programmer's *logical* interface to the machine.

References

[1] Anderson, B., *Type Syntax in the Language C: An Object Lesson in Syntactic Innovation*, in "Comparing and Assessing Programming Languages: Ada, C, and Pascal", Feuer & Gehani, eds., Prentice-Hall, 1984.

[2] Pohl, I. & Edelson, D., *A to Z: C Language Shortcomings*, Computer Languages, 13(2), pp. 51-64, Pergamon Press, 1988.

[3] Evans, A. *A Comparison of Programming Languages: Ada, Pascal and C*, in "Comparing and Assessing Programming Languages: Ada, C, and Pascal", Feuer & Gehani, eds., Prentice-Hall, 1984.

[4] Stroustrup, B., *The Design and Evolution of* C++, Section 9.4.4., Addison-Wesley, 1994.

[5] Murray, R., *A Statically Typed Abstract Representation for C++ Programs*, Proc. USENIX C++ Conference, Portland, Oregon, August 1992.

[6] Koenig, A., *Space Efficient Trees in* C++, Proc. USENIX C++ Conference, Portland, Oregon, August 1992.

[7] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language* C++, ANSI Document X3J16/95–0087.

[8] Coplien, J., *Advanced C++: Programming Styles and Idioms*, Section 7.4, Addison-Wesley, 1992.

Appendix A – Declaration of `set_new_handler`

The correct declaration of **`set_new_handler`**is:

```
void (*set_new_handler(void (*)(void)))(void);
```

This is so complicated that it is usually declared in two stages:

```
typedef void (*new_handler)(void);
new_handler set_new_handler(new_handler);
```

The use of a typedef is not, in our opinion, a substantial improvement.

The equivalent declarations in SPECS are considerably cleaner:

```
func set_new_handler : (^(void->void) -> ^(void->void));
```

and:

```
type new_handler : ^(void->void);
func set_new_handler : (new_handler -> new_handler);
```

Appendix B – Keywords [lex.key]

The following are SPECS keywords and cannot be used as identifiers or otherwise (those preceded by a bullet are new keywords, not inherited from C++) :

• **abstract**	• **dtor**	**long**	**static_cast**
auto	**dynamic_cast**	**mutable**	**switch**
• **bits**	**else**	**namespace**	**this**
bool	**enum**	**new**	**throw**
break	**explicit**	• **obj**	**true**
case	**extern**	**operator**	**try**
• **cast**	**false**	• **placement**	**type**
catch	**float**	• **post**	**typeid**
char	**for**	• **pre**	**typename**
class	**friend**	**private**	**union**
• **common**	• **func**	**protected**	**unsigned**
const	• **generate**	**public**	**using**
const_cast	**goto**	**register**	**virtual**
continue	**if**	**reinterpret_cast**	**void**
• **ctor**	• **inherits**	**return**	**volatile**
default	• **initially**	**short**	**wchar_t**
• **defined_cast**	**inline**	**signed**	**while**
delete	**int**	**sizeof**	
do	**label**	• **spec**	
double	• **lang**	**static**	

Appendix C – SPECS Example

The following is an example of SPECS code, provided so as to convey something of the "flavour" of the language. It is a line-for-line translation of the templated Stack class presented as Figure 7-1 in [8]. The output from the translator is in Appendix D.

```
type Cell<[type T]> : class
{
[friend]
     type Stack<[type T]> : class;
[private]
     obj next : ^Cell;
     obj rep  : ^T;
     func ctor  :  (r:^T,  c:^Cell<[T]>)
          initially  rep(r),  next(c)
          {}
}

type Stack<[type T]> : class
{
[public]
     func pop   : (void -> ^T);
     func top   : (void -> ^T)    { return rep^.rep; }
     func push  : (v:^T -> void)  { rep := new Cell<[T]>(v,rep); }
     func empty : (void -> int)   { return rep=0; }
     func ctor  : (void)          { rep := 0; }
[private]
     obj rep : ^Cell<[T]>;
}

func Stack<[type T]>::pop : (void -> ^T)
{
     obj ret(rep^.rep) : ^T;
     obj c(rep) : ^Cell<[T]>;
     rep := rep^.next;
     delete c;
     return rep;
}
```

Appendix D – Translation of SPECS Example

This is the output produced by the SPECS translator when given the SPECS code from Appendix C. It was produced in 'no line numbering' mode, and has not been "beautified" in any way, except for the removal of a few excess blank lines within the output. The indenting has not been modified at all.

```
template<class  T>
class  Cell  {
       template<class  T>
       friend  class  Stack;
private:
      Cell  *next;
      T  *rep;
       Cell(T  *r,  Cell<T>  *c)  :  rep(r),  next(c)  {}
};

template<class  T>
class  Stack  {
public:
      T  *pop(void);
      T  *top(void)  {
            return  rep->rep;
      }
      void  push(T  *v)  {
            rep  =  new  (Cell<T>)(v,  rep);
      }
      int  empty(void)  {
           return  rep  ==  0;
      }
      Stack(void)  {
           rep  =  0;
      }
private:
      Cell<T>  *rep;
};

template<class  T>
T  *Stack<T>::pop(void)  {
       T  *ret(rep->rep);
       Cell<T>  *c(rep);
       rep  =  rep->next;
      delete  c;
      return  rep;
}
```

Appendix E
Using The SPECS Translator

Use of the SPECS translator is very similar to use of the GNU g++ compiler. However, the executable is named **`specs`** and not **`g++`**, and the following command-line argument are changed:

`-EE`	output .cpp file (after preprocessing, before translation)
`-E`	output .ii file (after translation, before compiling)
`-noline`	don't generate line directives for g++
`-nousing`	convert using directives to access declarations where appropriate (use if compiler doesn't support 'using')
`-maxerrs=X`	maximum number of errors for SPECS to find
`-quiet`	don't print banner

Appendix F
SPECS Grammar Summary

F.1. Literals

```
Literal:
            INTEGER_CONSTANT
        |   CHARACTER_CONSTANT
        |   FLOATING_CONSTANT
        |   STRING_LITERAL
        |   BOOLEAN_CONSTANT
```

F.2. Names

```
ObjectName:
            ObjectNameCore
        |   :: ObjectNameCore

FuncName:
            FuncNameCore
        |   :: FuncNameCore
        |   OperatorFuncNameCore
        |   :: OperatorFuncNameCore

TypeName:
            TypeNameCore
        |   :: TypeNameCore

DtorName:
            dtor
        |   DtorNameCore
        |   :: DtorNameCore

CtorName:
            ctor
        |   CtorNameCore
        |   :: CtorNameCore

ResolvedName:
            ResolvedNameCore
        |   :: ResolvedNameCore

ResolvedPrefix:
            ResolvedPrefixCore
        |   :: ResolvedPrefixCore

IdentTemplArgs:
        IDENTIFIER  TemplateArgs

NameComponent:
            IDENTIFIER
        |   IdentTemplArgs
        |   IDENTIFIER  TemplateParams
        |   spec  TemplateParams  IdentTemplArgs
        |   ( spec  TemplateParams  IdentTemplArgs  )

ObjectNameCore:
            IDENTIFIER
        |   NameComponent  ::  ObjectNameCore

FuncNameCore:
            IDENTIFIER
        |   IdentTemplArgs
        |   IDENTIFIER  TemplateParams
        |   NameComponent  ::  FuncNameCore
```

```
TypeNameCore:
          IDENTIFIER
        | IdentTemplArgs
        | IDENTIFIER TemplateParams
        | spec TemplateParams IdentTemplArgs
        | ( spec TemplateParams IdentTemplArgs )
        | NameComponent :: TypeNameCore

OperatorFuncNameCore:
          OperatorID
        | defined_cast
        | NameComponent :: OperatorFuncNameCore

DtorNameCore:
          NameComponent :: dtor
        | NameComponent :: DtorNameCore

CtorNameCore:
          NameComponent :: ctor
        | NameComponent :: CtorNameCore

ResolvedNameCore:
          IDENTIFIER
        | IdentTemplArgs
        | IDENTIFIER :: ResolvedNameCore
        | IdentTemplArgs :: ResolvedNameCore

ResolvedPrefixCore:
          IDENTIFIER ::
        | IdentTemplArgs ::
        | IDENTIFIER :: ResolvedPrefixCore
        | IdentTemplArgs :: ResolvedPrefixCore
```

F.3. Expressions

```
PrimaryExpression:
          Literal
        | this
        | ( Expression )
        | IdExpression

IdExpression:
          ResolvedName
        | OperatorID
        | :: OperatorID
        | ResolvedPrefix OperatorID
        | ResolvedPrefix dtor
```

```
PostfixExpression:
            PrimaryExpression
        |   PostfixExpression ^
        |   PostfixExpression @
        |   PostfixExpression [ Expression ]
        |   PostfixExpression ( )
        |   PostfixExpression ( ExpressionList )
        |   FundamentalType ( ExpressionList )
        |   PostfixExpression . IdExpression
        |   PostfixExpression ^. IdExpression
        |   PostfixExpression . ( PostfixExpression ^ )
        |   PostfixExpression ^. ( PostfixExpression ^ )
        |   PostfixExpression ++
        |   PostfixExpression --
        |   dynamic_cast <[ TypeID ]> ( Expression )
        |   static_cast <[ TypeID ]> ( Expression )
        |   reinterpret_cast   <[ TypeID ]> ( Expression )
        |   const_cast <[ TypeID ]> ( Expression )
        |   cast <[ TypeID ]> ( Expression )
        |   typeid ( Expression )
        |   typeid ( RestrictedTypeID )

ExpressionList:
            AssignmentExpression
        |   ExpressionList , AssignmentExpression

UnaryExpression:
            PostfixExpression
        |   ++ UnaryExpression
        |   -- UnaryExpression
        |   UnaryOperator UnaryExpression
        |   sizeof UnaryExpression
        |   sizeof ( RestrictedTypeID )

UnaryOperator:
            +
        |   -
        |   !
        |   ~

NewExpression:
            new TypeID NewInitializer_opt NewPlacement_opt
        |   :: new TypeID NewInitializer_opt NewPlacement_opt

NewPlacement_opt:
            [empty]
        |   placement ( ExpressionList )

NewInitializer_opt:
            [empty]
        |   ( )
        |   ( ExpressionList )

DeleteExpression:
            delete DeleteParameters
        |   :: delete DeleteParameters
        |   delete [ ] DeleteParameters
        |   :: delete [ ] DeleteParameters

DeleteParameters:
            UnaryExpression
        |   UnaryExpression placement ( ExpressionList )

AllocationExpression:
            UnaryExpression
        |   NewExpression
        |   DeleteExpression
```

```
MultiplicativeExpression:
        AllocationExpression
      | MultiplicativeExpression  *  AllocationExpression
      | MultiplicativeExpression  /  AllocationExpression
      | MultiplicativeExpression  %  AllocationExpression

AdditiveExpression:
        MultiplicativeExpression
      | AdditiveExpression  +  MultiplicativeExpression
      | AdditiveExpression  -  MultiplicativeExpression

ShiftExpression:
        AdditiveExpression
      | ShiftExpression  <<  AdditiveExpression
      | ShiftExpression  >>  AdditiveExpression

RelationalExpression:
        ShiftExpression
      | RelationalExpression  <  ShiftExpression
      | RelationalExpression  >  ShiftExpression
      | RelationalExpression  <=  ShiftExpression
      | RelationalExpression  >=  ShiftExpression

EqualityExpression:
        RelationalExpression
      | EqualityExpression  =  RelationalExpression
      | EqualityExpression  !=  RelationalExpression

AndExpression:
        EqualityExpression
      | AndExpression  &  EqualityExpression

ExclusiveOrExpression:
        AndExpression
      | ExclusiveOrExpression  !  AndExpression

InclusiveOrExpression:
        ExclusiveOrExpression
      | InclusiveOrExpression  |    ExclusiveOrExpression

LogicalAndExpression:
        InclusiveOrExpression
      | LogicalAndExpression  &&  InclusiveOrExpression

LogicalOrExpression:
        LogicalAndExpression
      | LogicalOrExpression  ||  LogicalAndExpression

ConditionalExpression:
        LogicalOrExpression
      | LogicalOrExpression  ?  Expression  :  AssignmentExpression

AssignmentExpression:
        ConditionalExpression
      | AllocationExpression  AssignmentOperator  AssignmentExpression
      | ThrowExpression
```

AssignmentOperator:
```
    :=
  | *:=
  | /:=
  | %:=
  | +:=
  | -:=
  | >>:=
  | <<:=
  | &:=
  | |:=
  | !:=
```

Expression:
 AssignmentExpression
 | *Expression* **,** *AssignmentExpression*

ConstantExpression:
 ConditionalExpression

F.4. Statements

Statement:
 ExpressionStatement
 | *CompoundStatement*
 | *SelectionStatement*
 | *IterationStatement*
 | *JumpStatement*
 | *DeclarationStatement*
 | *TryBlock*

ExpressionStatement:
 ;
 | *Expression* **;**

CompoundStatement:
 { }
 | **{** *StatementSeq* **}**

StatementSeq:
 Statement
 | *StatementSeq* *Statement*

SelectionStatement:
 if (*Condition* **)** *CompoundStatement* *SelectionElse*
 | **switch (** *Condition* **) {** *SwitchCore* **}**

SelectionElse:
 [empty]
 | **else if (** *Condition* **)** *CompoundStatement* *SelectionElse*
 | **else** *CompoundStatement*

SwitchValues:
 ConstantExpression
 | *SwitchValues* **,** *ConstantExpression*

SwitchCore:
 [empty]
 | *SwitchCore* **case** *SwitchValues* **:** *CompoundStatement*
 | *SwitchCore* **default :** *CompoundStatement*

Condition:
 Expression
 | *ObjKeyword* *ObjectName* **:=** *AssignmentExpression* **:** *ObjectSpecifiers* *TypeID*
 | *ObjKeyword* **(** *ObjectName* **:=** *AssignmentExpression* **) :** *ObjectSpecifiers* *TypeID*

Condition_opt:
[empty]
| *Condition*

IterationStatement:
while **(** *Condition* **)** *CompoundStatement*
| **do** *CompoundStatement* **while** **(** *Expression* **)** **;**
| **for** **(** *ForInitStatement* *Condition_opt* **;** **)** *CompoundStatement*
| **for** **(** *ForInitStatement* *Condition_opt* **;** *Expression* **)** *CompoundStatement*

ForInitStatement:
ExpressionStatement
| *ObjKeyword* *ObjectPrefix* *TypeSpec_Simple*

JumpStatement:
break **;**
| **continue** **;**
| **return** **;**
| **return** *Expression* **;**
| **goto** *IDENTIFIER* **;**
| **label** *IDENTIFIER* **:**

DeclarationStatement:
BlockDeclaration

F.5. Declarations

DeclarationSeq:
[empty]
| *DeclarationSeq* *Declaration*

Declaration:
BlockDeclaration
| *FunctionDefinition*
| *LinkageSpecifier*
| *NamespaceDefinition*

BlockDeclaration:
AsmSpecifier
| *TypeDeclaration*
| *ObjectDeclaration*
| *FunctionDeclaration*
| *NamespaceAliasDefinition*
| *UsingDeclaration*
| *UsingDirective*

ObjKeyword:
obj

TypeKeyword:
type

FuncKeyword:
func

FunctionPrefix:
FuncKeyword *FuncName* **:** *FunctionSpecifiers* *FuncDeclarator*

FunctionDefinition:
FunctionPrefix *FuncBody*
| *FunctionPrefix* **try** *FuncBody* *HandlerSeq*
| *FuncKeyword* *CtorDtorDefinition*

FunctionDeclaration:
FunctionPrefix *Generate_opt* **;**
| *FuncKeyword* *CtorDtorDeclaration*

```
ObjectDeclaration:
        ObjKeyword  ObjectPrefix  TypeSpec_Class
      | ObjKeyword  ObjectPrefix  TypeSpec_Union
      | ObjKeyword  ObjectPrefix  TypeSpec_Simple
      | ObjKeyword  ObjectPrefix  TypeSpec_Enum

TypePrefix:
      TypeKeyword  TypeName  :

TypeDeclaration:
        TypePrefix  TypeSpec_Class
      | TypePrefix  TypeSpec_Union
      | TypeKeyword  :  TypeSpec_Union
      | TypePrefix  TypeSpec_Simple
      | TypePrefix  TypeSpec_Enum
      | TypeKeyword  :  TypeSpec_Enum

NamespaceDefinition:
        namespace  {  NamespaceBody  }
      | namespace  IDENTIFIER  {  NamespaceBody  }

LinkageSpecifier:
      lang  STRING_LITERAL  {  LINKAGE_CONTENTS  }

AsmSpecifier:
      lang  “asm”  {  AsmDirectives  }

AsmDirectives:
        [empty]
      | AsmDirectives  ASM_DIRECTIVE
```

F.6. Namespace Declarations

```
NamespaceBody:
      DeclarationSeq

NamespaceAliasDefinition:
      namespace  IDENTIFIER  :=  QualifiedNamespaceSpecifier  ;

QualifiedNamespaceSpecifier:
      ResolvedName

UsingDeclaration:
        using  type  ResolvedName  ;
      | using  obj  ResolvedName  ;
      | using  func  IdExpression  ;

UsingDirective:
      using  namespace  ResolvedName  ;
```

F.7. Object Declarations

```
ObjectDeclarator:
        ObjectName
      | (  ObjectName  )
      | ObjectName  :=  ObjInitValue
      | (  ObjectName  :=  ObjInitValue  )
      | ObjectName  ObjConstructor
      | (  ObjectName  ObjConstructor  )

ObjectPrefix:
        ObjectPrefixMain
      | :  ObjectSpecifiers
```

ObjectPrefixMain:
    ObjectDeclarator **:** *ObjectSpecifiers*
  | *ObjectDeclarator* **,** *ObjectPrefixMain*

ObjInitList:
    ObjInitValue
  | *ObjInitList* **,** *ObjInitValue*

ObjInitValue:
    AssignmentExpression
  | **{** *ObjInitList* **}**
  | **{ }**

ObjConstructor:
    ( )
  | **(** *ExpressionList* **)**

ObjectSpecifier:
    common
  | **mutable**
  | **auto**
  | **register**
  | **static**
  | **extern**
  | **bits (** *ConstantExpression* **)**

ObjectSpecifiers:
    [empty]
  | *ObjectSpecifiers ObjectSpecifier*

F.8. Type Declarations

TypeSpec_Class:
    class *Generate_opt* **;**
  | **class {** *ClassBody* **}**
  | **class** *DerivedList* **{** *ClassBody* **}**

TypeSpec_Union:
    union ;
  | **union {** *ClassBody* **}**

TypeSpec_Simple:
  TypeID **;**

TypeSpec_Enum:
  enum { *EnumList_opt* **}**

EnumList:
    Enumerator
  | *EnumList* **,** *Enumerator*

EnumList_opt:
    [empty]
  | *EnumList*

Enumerator:
    IDENTIFIER
  | *IDENTIFIER* **:=** *ConstantExpression*

F.9. Function Declarations

ParameterPrefix:
    IDENTIFIER **:**
  | *IDENTIFIER* **:=** *AssignmentExpression* **:**

Parameter:
    ParameterPrefix TypeID

FunctionParameter:
      Parameter
    | **`obj`** *Parameter*
    | *TypeID*

FuncParameterListCore:
      FunctionParameter
    | *FuncParameterListCore* **`,`** *FunctionParameter*

FuncParameterList:
      `...`
    | *FuncParameterListCore* **`,`** *FunctionParameter*
    | *FuncParameterListCore* **`,`** **`...`**

FuncParameters:
      Parameter
    | **`(`** *Parameter* **`)`**
    | **`obj`** *Parameter*
    | **`(`** **`obj`** *Parameter* **`)`**
    | *TypeID*
    | *FuncParameterList*
    | **`(`** *FuncParameterList* **`)`**

FuncDeclarator:
      `(` *FuncParameters* **`->`** *TypeID* **`)`** *ExceptionSpecification_opt*
    | *CVQualifiers* **`(`** *FuncParameters* **`->`** *TypeID* **`)`** *ExceptionSpecification_opt*

FuncBody:
    CompoundStatement

FunctionSpecifier:
      `inline`
    | **`virtual`**
    | **`abstract`**
    | **`explicit`**
    | **`common`**
    | **`static`**
    | **`extern`**

FunctionSpecifiers:
      [empty]
    | *FunctionSpecifiers FunctionSpecifier*

F.10. Declarators

SizeSpecifier:
      `short`
    | **`long`**

SignSpecifier:
      `signed`
    | **`unsigned`**

SizeAndSignSpecifiers:
      SizeSpecifier SignSpecifier
    | *SignSpecifier SizeSpecifier*

```
FundamentalType:
          char
        | SignSpecifier char
        | wchar_t
        | bool
        | int
        | SignSpecifier int
        | SizeSpecifier int
        | SizeAndSignSpecifiers int
        | float
        | double
        | long double
        | void

TypeID:
          RestrictedTypeID
        | BracketedName

BracketedName:
          ResolvedName
        | ( FuncParameters -> TypeID ) ExceptionSpecification_opt
        | ( BracketedName )

RestrictedTypeID:
          FundamentalType
        | CVQualifiers FundamentalType
        | TypeIDKeyword ResolvedName
        | CVQualifiers TypeIDKeyword ResolvedName
        | CVQualifiers ResolvedName
        | ( RestrictedTypeID )
        | ArrayDeclarator TypeID
        | PointerDeclarator TypeID
        | CVQualifiers PointerDeclarator TypeID

TypeIDKeyword:
          typename
        | class
        | union
        | enum

PointerDeclarator:
          ^
        | ResolvedPrefix ^
        | &

ArrayDeclarator:
          [ ]
        | [ ConstantExpression ]

CVQualifier:
          const
        | volatile

CVQualifiers:
          CVQualifier
        | CVQualifiers CVQualifier
```

F.11. Classes

```
ClassBody:
          [empty]
        | ClassBody [ AccessSpecifier ]
        | ClassBody [ friend ]
        | ClassBody BlockDeclaration
        | ClassBody FunctionDefinition
```

```
AccessSpecifier:
        private
      | protected
      | public

DerivedList:
      inherits DerivedListContents

DerivedListContents:
        DerivedListItem
      | DerivedListContents , DerivedListItem

DerivedListItem:
        ResolvedName
      | AccessSpecifier ResolvedName
      | virtual ResolvedName
      | virtual AccessSpecifier ResolvedName
      | AccessSpecifier virtual ResolvedName

CtorDtorDefinition:
        CtorPrefix CtorBody
      | DtorPrefix DtorBody

CtorDtorDeclaration:
        CtorPrefix Generate_opt ;
      | DtorPrefix Generate_opt ;

CtorPrefix:
        CtorName : FunctionSpecifiers CtorParameters
      | CtorName TemplateParams : FunctionSpecifiers CtorParameters

DtorPrefix:
      DtorName : FunctionSpecifiers DtorParameters

CtorParameters:
      ( FuncParameters ) ExceptionSpecification_opt

DtorParameters:
      ( void ) ExceptionSpecification_opt

CtorBody:
        FuncBody
      | initially CtorInitializerList FuncBody
      | try FuncBody HandlerSeq
      | try initially CtorInitializerList FuncBody HandlerSeq

DtorBody:
        FuncBody
      | try FuncBody HandlerSeq

CtorInitializerList:
        MemberInit
      | CtorInitializerList , MemberInit

MemberInit:
        ResolvedName ObjConstructor
      | IDENTIFIER :: ctor ObjConstructor
      | IdentTemplArgs :: ctor ObjConstructor
```

F.12. Operators

```
OperatorID:
        operator Operator

Operator:
        new
        | delete
        | new[]
        | delete[]
        | +
        | -
        | *
        | /
        | %
        | ^
        | &
        | &
        | |
        | ~
        | !
        | :=
        | <
        | >
        | +:=
        | -:=
        | *:=
        | /:=
        | %:=
        | !:=
        | &:=
        | |:=
        | <<
        | >>
        | <<:=
        | >>:=
        | =
        | !=
        | <=
        | >=
        | &&
        | ||
        | PreOrPost ++
        | PreOrPost --
        | ,
        | ^.^
        | ^.
        | ()
        | []

PreOrPost:
          pre
        | post
```

F.13. Templates

```
TemplateParamDeclarator:
          obj Parameter
        | type TemplateParamType
        | type TemplateParamType := TypeID

TemplateParamType:
          IDENTIFIER
        | IDENTIFIER TemplateParams

TemplateParamList:
          TemplateParamDeclarator
        | TemplateParamList , TemplateParamDeclarator

TemplateParams:
        <[ TemplateParamList ]>

TemplateArgTerm:
          RestrictedTypeID
        | AssignmentExpression
```

```
TemplateArgList:
        TemplateArgTerm
      | TemplateArgList  ,  TemplateArgTerm

TemplateArgList_opt:
        [empty]
      | TemplateArgList

TemplateArgs:
      <[  TemplateArgList_opt  ]>

Generate_opt:
        [empty]
      | generate
```

F.14. Exceptions

```
TryBlock:
      try CompoundStatement  HandlerSeq

HandlerSeq:
        Handler
      | HandlerSeq  Handler

Handler:
      catch  (  ExceptionDeclaration  )  CompoundStatement

ExceptionDeclaration:
        FunctionParameter
      | ...

ThrowExpression:
        throw
      | throw  AssignmentExpression

ExceptionSpecification_opt:
        [empty]
      | throw  (  )
      | throw  (  TypeIDList  )

TypeIDList:
        TypeID
      | TypeIDList  ,  TypeID
```