APPLICATION BINARY INTERFACE COMPATIBILITY THROUGH A CUSTOMIZABLE LANGUAGE

by

Kevin Jay Atkinson

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2011

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Kevin Jay Atkinson**

has been approved by the following supervisory committee members:

Matthew Flatt	, Chair	**11/3/2011** Date Approved
Gary Lindstrom	, Member	**11/17/2011** Date Approved
Eric Eide	, Member	**11/3/2011** Date Approved
Robert Kessler	, Member	**11/3/2011** Date Approved
Olin Shivers	, Member	**11/29/2011** Date Approved

and by **Al Davis**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

ZL is a C++-compatible language in which high-level constructs, such as classes, are defined using macros over a C-like core language. This approach is similar in spirit to Scheme and makes many parts of the language easily customizable. For example, since the class construct can be defined using macros, a programmer can have complete control over the memory layout of objects. Using this capability, a programmer can mitigate certain problems in software evolution such as fragile ABIs (Application Binary Interfaces) due to software changes and incompatible ABIs due to compiler changes.

ZL's parser and macro expander is similar to that of Scheme. Unlike Scheme, however, ZL must deal with C's richer syntax. Specifically, support for context-sensitive parsing and multiple syntactic categories (expressions, statements, types, etc.) leads to novel strategies for parsing and macro expansion.

In this dissertation we describe ZL's approach to parsing and macros. We demonstrate how to use ZL to avoid problems with ABI instability through techniques such as fixing the size of class instances and controlling the layout of virtual method dispatch tables. We also demonstrate how to avoid problems with ABI incompatibility by implementing another compiler's ABI.

Future work includes a more complete implementation of C++ and elevating the approach so that it is driven by a declarative ABI specification language.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my co-advisers Matthew Flatt and Gary Lindstrom—who were also coauthors for works that are part of this dissertation—for their support and contributions to this dissertation. I would also like to thank my other committee members, Eric Eide, Bob Kessler, and Olin Shivers for their support and feedback on this dissertation.

In addition I would like to thank Ryan Culpepper, Carl Eastlund, and Jon Rafkind for feedback on works that are part of this dissertation.

I also want to thank Jay Lepreau (though he passed away) and Eric Eide for their financial support through the Flux Research Group.

This work is based on an earlier work [12]: "ABI Compatibility Through a Customizable Language", Proceedings of the *Ninth International Conference on Generative Programming and Component Engineering (GPCE'10)*, Eindhoven, The Netherlands, Oct. 2010. © ACM, 2010, `http://dx.doi.org/10.1145/1868294.1868316`.

Parts of this dissertation also appear in the work [11]: "Adapting Scheme-Like Macros to a C-Like Language", *Workshop on Scheme and Functional Programming*, Portland, Oregon, Oct. 2011.

CHAPTER 1

INTRODUCTION

There are two types of programming interfaces to a library: the *Application Programming Interface* (API) and the *Application Binary Interface* (ABI). The API defines the ways a programmer may request services from the library. Some of the constituents of an API in an object-oriented language are the names of classes, the methods they support, and the types of the arguments that methods take. What goes into the API is under the control of the library designer. An ABI is the object-code equivalent of an API. It is the low-level interface between the application and the library. A compiler implements a mapping from a library's API to its ABI. Some of the constituents of the mapping include calling conventions and class layout. Unlike the API, the programmer has little to no control of the ABI in most languages.

When a library designer changes an API in a way that preserves backwards compatibility with previous releases, *source code compatibility* is maintained. That is, existing applications that use a library do not need to change at the source level. However, even if source code compatibility is preserved, *binary compatibility* need not be preserved; existing applications may need to be recompiled because the compiler typically does not guarantee ABI compatibility with API compatibility.

In situations when a library is used by a small number of programs that can easily be recompiled, breaking binary compatibility between releases may be acceptable. However, if a large number of programs depend on the library, then recompiling is not an acceptable option as it can take anywhere from hours to days to recompile everything. In addition, in many situations the source code for applications using the library is not available, thus making upgrading impossible unless binary compatibility is preserved.

Preserving binary compatibility for C++ programs is difficult because the typical C++ ABI is extremely fragile. Seemingly simple changes, such as adding methods, may break

binary compatibility. In fact, almost any change to a class declaration will likely break binary compatibility and require applications that use the library to be recompiled.

In addition, the C++ ABI is not well defined as every compiler implements the C++ standard in a slightly different way. Libraries compiled with one compiler, such as Visual C++, generally will not be usable by applications compiled with a different compiler, such as GCC. Furthermore, the ABI may change between releases of the same compiler. Thus, upgrading to a newer compiler may also break binary compatibility.

In contrast to C++, the C ABI is simple and well defined for a given architecture and operating system. Since the C ABI is far simpler than the C++ ABI, preserving binary compatibility is much easier. Furthermore, since the C ABI is well defined for a given architecture, compatibility between compilers is a nonissue. In fact, some C++ applications export only a C API for these very reasons.

The C ABI is successful because of its simplicity and consistency. That simplicity, in turn, is based in part on the simplicity of the C language. As languages become more complicated, so do the number of choices to be made in an ABI. Thus, ABIs for complicated languages, such as C++, tend to vary among compilers and even among versions of a compiler. Standardizing on one C++ ABI would solve the incompatibility problem. Although some effort has been made in that area with Itanium C++ ABI [7], there are still several C++ ABIs in common use, most notably the GCC and Visual C++ ABIs.

Even if all C++ compilers standardized on a single ABI, the problem of preserving binary compatibility between releases of a library would still be a major problem. This is because most C++ ABIs, including the Itanium C++ ABI, are optimized for performance, not for preserving binary compatibility. Previous designs for a less fragile ABI for C++ [48, 38] make significant sacrifices in performance. Thus, library designers must make a choice between breaking binary compatibility between releases or contorting their programs to preserve it by using a variety of programming idioms.

We could try to add a few extensions to C++, such as a choice of different ABIs or support for common programming idioms, but a fixed number of extensions will never be enough as the problem of preserving binary compatibility is far too complex. A

nonextensible language cannot and should not support every possible rarely needed case. A more general and integrated approach is an extensible compiler. Traditional extensible compiler designs treat a compiler extension as an entity separate from the code to be compiled. On the other hand, a *macro system* acts as an extensible compiler and also allows the programmer to implement code and compiler extensions together, thus elevating compiler extensions to the level of a library. This, in turn, allows different ABI choices to be incorporated with different parts of an application. For example, one class can use an ABI optimized for performance while another uses an ABI aimed at preserving binary compatibility.

A simple macro system, such as the C preprocessor, is not adequate for defining compiler extensions. Rather, the macro system must be an integral part of the language, and it must be able to do more than simply rearrange syntax. In addition to providing macro primitives, a language for giving the programmer control over an ABI must include a carefully designed core that allows higher-level constructs, such as classes, to be implemented via macros. This capacity enables the programmer to redefine key aspects that affect ABI attributes, such as class layout. ZL, a C++ compatible systems programming language that is the subject of this dissertation, does exactly this.

For relatively simple language extensions ZL supports pattern-based macros similar to Scheme's `syntax-rules` [51]. In addition, ZL supports parser extensions that change the tokenization (roughly) of the input source, so that macro uses need not have the restricted form that Scheme's macro system imposes. Even with such extensions, pattern-based macros are limited. Therefore, in the same way that Scheme provides procedural macros via `syntax-case` [24], ZL supports procedural macros. ZL's API for procedural macros includes support for reflective tasks such as getting the value of a macro parameter, determining whether a symbol is currently defined and getting basic properties about the symbol, and other necessary tasks to implement a class system.

1.1 Dissertation Statement

Fragile and incompatible ABIs are a major problem in software maintenance and evolution that can be systematically dealt with and effectively managed using a macro-based

system that allows the programmer to control how an API maps to an ABI.

1.2 Approach

This dissertation demonstrates the thesis in the context of C++, through the use of ZL. Class layout is a key aspect of the C++ ABI and is hence the focus of our research. However, we support other parts of the ABI as well. For example, we support name mangling, which is how local symbol names are translated in order to make them globally unique.

Although, the ZL language gives the user complete control over many things that affect the ABI, since our implementation of ZL compiles to a C like language, it does not give the user control over everything such as exceptions and calling conventions. This is not a problem, however, since exception support is beyond the scope of our research and calling conventions are stable within a given architecture.

In addition, C++ is a very complicated language and this research only addressed the features of C++ most relevant to the research question. In particular we did not address multiple-inheritance, exceptions, and templates, which pose unique challenges.

1.3 Contributions

Our contributions in this dissertation are two-fold. The first is to demonstrate how ZL can be used to mitigate the problem of binary incompatibility through the use of macros. The second is to demonstrate the adaptation of Scheme-style, hygienic macros to C-style syntax.

This dissertation outlines the problems of binary compatibility in C++ (in Chapter 2) and shows how a macro system can help (in Chapter 3). We then, after giving an overview of ZL (in Chapter 4), demonstrate how ZL can be used to mitigate the problem of binary incompatibility (in Chapters 5 and 6). For example, we show how to avoid breaking binary compatibility when adding new data members or methods to a class. We also match GCC's ABI to the point where a simple library can be compiled with ZL and then used with GCC and vice versa. In addition to ZL's native ABI and GCC's ABI, we implement several other specialized ABIs and show how classes with different ABIs can be used in the same program.

This dissertation also presents the details of ZL parser and macro expander (in Chapters 4, 7, and 8). Dealing with C's idiosyncratic syntax introduces complexities that are not solved by simply converting the original text into an S-expression intermediate format. Instead, parsing of raw text must be interleaved with the expansion process, and hygiene rules must be adapted carefully to actions such as accessing structure members.

CHAPTER 2

PROBLEMS WITH THE C++ ABI

This chapter outlines what goes into the C++ ABI, why it is so fragile, and the problems both the fragility and being compiler specific cause.

2.1 The C++ ABI

There are many components to the C++ ABI. Of them, the components of most interest to this dissertation are:

- *Data Layout.* An ABI specifies how data are laid out in a memory region representing an instance (struct) of a class. These data include the data members of the class but it may also include other auxiliary information needed by the compiler such as a pointer to the virtual table (vtable). If a struct only contains data members and nonvirtual functions, and does not inherit from any other classes, it is generally considered a POD (plain old data) datatype. The layout of POD objects is the same as it would be in the C API. If the structure or class is not a POD data type than the layout is essentially left undefined by the C++ standard. However, in general a pointer to the virtual table is included first, then the data-members of any nonvirtual base classes, or a pointer to the class in the case of a virtual base class, then finally the data members of the current class.

- *Virtual Table Layout.* A virtual table (vtable) is the table that is used to dispatch virtual functions and contain run-time type information (RTTI), among other things. A vtable is not part of a C++ standard but it is included in nearly every C++ ABI. The virtual table is generally a static object that is included in the object file and then copied into memory at load time. A virtual table generally includes the following items:

  - The typeinfo pointer for RTTI
  - The displacement to the top of the object from the location within the outer object.
  - Virtual function pointers, which are used for virtual function dispatch
  - Copies of the virtual tables for any nonvirtual base classes
  - Pointers to virtual tables for any virtual base classes

- *Construction and Destruction.* An ABI defines how objects are created and destroyed. In C++ this is done via special member functions known as constructors and destructors. These functions are created automatically by the compiler, but the user can control part of the contents. What is involved in object construction is part of the ABI specification. An ABI, such as GCC's [7], may even emit multiple versions of the same constructor or destructor.

- *New and Delete.* An ABI defines how `new`, `delete`, and `delete []` are implemented. Often these operators call ABI specific functions rather than just calling C's `malloc` and `free`.

- *Name Mangling.* An ABI specifies how a function's local symbol names are mangled in order to make them globally unique. Some of the things that go into the mangled name include 1) the local name of the symbol, 2) the types of the parameters for functions, 3) the class name for member function, and 4) the namespace the symbol is in.

Other important components include:

- *RTTI.* In addition to information to implement inheritance, an ABI also contains some run-time type information. In C++ the RTTI has three purposes: 1) to support the typeid operation, 2) to match an exception handler with a thrown object, and 3) to implement the dynamic_cast operator [7, §2.9].

- *Calling Conventions.* Calling conventions for nonmember functions are the same as they are in the C ABI. Calling conventions for nonvirtual member functions

are generally the same as for nonmember functions with the first parameter being the `this` pointer. However, this is not always the case. For example, Microsoft Visual C++ passes the `this` pointer in a register, rather than passing it is as the first parameter. The calling convention for virtual functions involves a lookup in the vtable and thus varies from one ABI to another.

- *Exception Handling.* An ABI also specifies how exceptions are handled. There are many different possible ways to implement exceptions.

- *Layout of The Object File.* The layout of the object file is generally not part of the C++ ABI specification. It is generally left to other standards such as ELF.

- *Linkage.* Symbol lookup in C++ is generally delegated to the standard linker for the platform. However, unlike with C, many objects in C++ are not clearly part of any single object file. Examples include:

  - Out-of-line Functions
  - Static Data
  - Virtual Tables
  - Typeinfo
  - Constructors and Destructor
  - Instantiated Templates

  These symbols can thus appear in multiple object files. As a result, the C++ compiler needs a way to inform the linker of these special symbols and the linker needs a policy to handle them.

- *Templates.* Templates are a large part of the C++ language but a small part of the ABI. As far as the C++ ABI is concerned templates are ordinary objects except for the fact that 1) the symbol names need to be mangled such that they include the template parameters, and 2) the same instantiation can appear in multiple object files and should be combined to save space when linking them together.

- Keeping track of the size of dynamically allocated arrays.

- Pointer to member functions. For nonvirtual functions this is generally a function pointer. For virtual functions it is the pointer into the virtual table.

2.2 The Problem of Fragile ABIs

The C++ ABI is extremely fragile as seemingly simple changes, such as adding data members or virtual methods to a class, may break binary compatibility. Adding data members breaks binary compatibility because it changes the size of the class, which is used at compile time when allocating objects on a stack or inlining one object in another. Similarly, adding virtual methods changes the size of the vtable, and thus, with most ABIs, changes the offsets of all the methods' function pointers for any subclasses. In fact, most changes to a class will break binary compatibility since they change the object's (or vtable's) layout in one way or another.

Due to the extremely fragility of the C++ ABI programmers go to great lengths to avoid breaking the ABI. For example many large software engineering projects have guidelines to that deal with this issue. Examples include KDE [27], BE [49], and Windows.

In fact the problem of fragile ABIs was a key consideration when developing the Java ABI as *The Java Language Specification* [39] has an entire chapter devoted to the issue of Binary Compatibility. The importance of binary compatibility was also recognized in the paper by Forman, et al., which they summarize as "Only application alteration necessitates recompilation" on page 430 [36] . According to Yu, et al. this paper was a precursor to to Java's concept of binary compatibility [58].

2.2.1 Solutions Within C++

For compiled languages like C++ there is no comprehensive solution to this problem; consequently, developers employ a large number of techniques to get around it.

One solution is to only export abstract base classes (ABCs), or interfaces, which will never change. When it is necessary to add new methods a *new* ABC is created. This technique is often accomplished by including version numbers in the name of the ABC. A very similar technique is used by the Microsoft Component Object Model [57]. This however, requires full encapsulation, i.e., no direct access to data members.

Another solution is to create a C API on top of the C++ one for the sole purpose of more easily maintaining binary compatibility. C ABIs are inherently less fragile than C++ since they are simpler. For example, we did this with the Aspell project [1].

Yet another solution is to be aware of what exactly will break an ABI and employ techniques to avoid doing so, many of which require planning ahead. For example, a dummy variable can reserve space ahead of time to avoid changing the size of an object when adding new data members. However, these techniques, which will be explored in detail in Chapter 3, can often lead to less maintainable code.

2.2.2 Defining a Better ABI

The main reason ABIs are so fragile for compiled languages such as C++ is that offsets and sizes are fixed at compile time. If this information were resolved at load-time then the issue of fragile ABIs would be greatly reduced.

Java does just that, by making nearly all references in the compiled Java bytecode symbolic. That is, not only are functions symbolic as they are in C, but so are calls to virtual methods; even data member lookups are symbolic. All of these symbols are resolved when the class is loaded. The Java Language Standard [39] is very careful to define the ABI in such a way that breaking binary compatibility will almost certainly mean breaking source code compatibility also.

Resolving any sort of detail that will affect ABI compatibility at load time is possible in Java since Java uses a completely different notion of compilation. In particular Java is compiled to byte code, not object code. This allows more flexibility in the type of information that can be resolved a load time.

Even if it is not practical to resolve everything that can possibly affect ABI compatibly at run time, the ABI can still be defined in such as way to make it significantly less fragile. Such an approach is done by Goldstein and Sloan [38]. They define a special ABI known as the Object Binary Interface which will only be used on request. The ABI they define allows for evolutionary steps such as adding new public and protected methods, and adding or removing private data members. However, it does not allow for changing the order or type of public data members. Thus it greatly reduces the problem of a fragile ABI but does not eliminate it. Also, their ABI is not without cost when compared to the more

traditional C++ ABI. Thus, it is likely to affect performance, especially since all inheritance is implemented in a manner similar to how virtual inheritance is implemented in traditional C++ ABIs.

2.3 The Problem of Compiler Specific ABIs

Due to the complexity of the C++ ABI, the implementation is compiler specific. Hence, changing compilers can also break binary compatibility. Thus, when using C++ libraries, not only is the specific version of the library important, but so is the compiler used to compile it.

The fact that a C++ library is tied to a particular ABI implementation is a particular problem in Windows when a large amount of code written is using Microsoft's VC++. Because so much code is written using VC++ in Windows many other Windows compilers often conform to at least part of this ABI, making it less of an issue. But it is still an issue since this ABI is not universally used by all C++ compilers. For example GCC uses a different ABI. Thus it is impossible to use GCC when developing Windows code that uses Windows C++ libraries. We ran into this problem a while ago, when we wanted to write some filters for AviSynth, a program for scriptable video processing. The source code for AviSynth is freely available but it will only compile on VC++. We wanted to write the filters using GCC. So in order to do this we had to write a special filter whose sole purpose was to bridge the gap by using a more stable C (as opposed to C++) interface. We then used this filter to write filters that could be compiled using GCC. These filters were written in C++. Thus we had to write a special "C" interface in order to interface with "C++" code, which seems silly, but was necessary since VC++ and GCC C++ ABIs are incompatible.

There is no real solution to the problem of incompatible ABIs. The general solution is to simply avoid the issue by just using a compiler that is compatible with the C++ ABI deployed. When this is not an option, then the only other solution is to write a C API as was done for AviSynth.

CHAPTER 3

SOLVING ABI PROBLEMS

Table 3.1 lists changes that can break binary compatibility without affecting source code compatibility. Except for the last item, this list is from the paper by Forman, et al. [36]. This chapter discusses each of the problems, the solutions used in practice, and how a macro system can improve on them.

3.1 Overview

For each ABI compatibility problem there are often several different solutions, and which one to choose depends on the situation. If there were one really good solution to the particular problem then it could easily be added as an extension to the language. In fact, since C++ is a fairly mature language, there is a good chance that the solution already

Table 3.1. Changes that can affect the ABI. Solutions to many of these problems can be supported to some extent within the constraints of the existing C++ ABI, but for some the only solution is a better ABI.

Change	Can Support	Section
add instance variable	Yes	3.2
add new method	Yes	3.3
reorder methods	Yes	3.4
reorder instance variables	Yes	3.4
remove private method	Yes	3.5
remove private instance variable	Yes	3.5
migrate method upward in class hierarchy	Yes	3.6
add parameters	Yes	3.7
insert new class in class hierarchy	New ABI	3.8
migrate parent downward in class hierarchy	New ABI	3.8
change compilers	-	3.10

would have been added. However, since there are many solutions, with none of them being clearly better than the other, language designers would have to add them all in order to deal with the problem. Adding all solutions is not an attractive option, as it would severely, possibly unnecessarily, bloat the language. Furthermore, there may be additional creative solutions, specific to a particular problem that language designers cannot possibly think of. Consequently it is essential to give the programmer as much control as practical for implementing the best solution for a given situation. Giving programmers control is a task ideally suited to a macro system where the implementation of the classes and other key parts of the ABI are under the programmer's control.

3.1.1 User Roles

A good macro system can benefit all users, but not everyone needs to know the full details of how macros work. There are three primary classes of users: 1) *End Users or Library Consumers*, who just use the library, but can benefit from increased binary compatibility; 2) *Library Implementers*, who can use the macro libraries to provide increased binary comparability, but do not need to know the details of the macro libraries themselves; and 3) *Tool Implementers*, who provide the macro libraries for the library implementers.

With traditional compiler designs, tool implementers are in relatively short supply, and they face a daunting task on two fronts: they must modify the compiler, and they must convince users of the library to use the modified compiler. Our approach to improving ABI compatibility is to simplify the tool implementer's job, so that library implementers will have better tools and end users will have more compatible libraries. Specifically, with a macro-extensible compiler that can express ABI details through the macro layer, tool implementers gain a simpler framework for implementing more interoperable designs, and they get a more composable framework so that multiple tools can be combined. In this way, a tool becomes more like a library.

Indeed, just as library consumers can become library implementers when they want to generalize their application code so that others can use it, library implementers can become tool implementers when they need to do something unusual for which a macro library does not yet exist. The key benefit of a macro system in this case is that it allows a library implementer to easily become a tool implementer.

3.2 Adding Private Data Members

A C++ ABI can change by adding new private data members since that changes the size of the object. For example, changing

```
class X {
private:
  int old_var;
public:
  ...
};
```

to

```
class X {
private:
  int old_var;
  int new_var;
public:
  ...
};
```

breaks binary compatibility, because the size of X changes from the size of one integer to the size of two integers. This change is a problem when the size of the object is needed at compile time, such as when the object is allocated directly on the stack, embedded inside another object, or even allocated with `new`.

3.2.1 Reserving Space Ahead of Time

One solution in C++ is to use a dummy variable to reserve space ahead of time. For example, changing X to:

```
class X {
  int old_var;
  unsigned long _reserved[3];
};
```

reserves enough space for three additional variables. Then to add a new variables simply decrease the size of `_reserved`:

```
class X {
  int old_var;
  int new_var;
  unsigned long _reserved[2];
};
```

This approach works but requires a bit of planning ahead. It also depends on knowing the size of the members, which varies amoung architectures. The above example relies on the fact that `long` is the same size as an `int`, which is not always the case. For example, on 64-bit processors, an `int` is 4 bytes while a `long` is 8. However, `long` is used, as opposed to an `int`, since on most architectures a `long` is the same size as a pointer.

Things get interesting when we run out of space. To deal with this situation, the last reserved slot is used as a pointer. For example:

```
class X {
private:
  int old_var;
  unsigned long _reserved[3];
public:
  ...
};
```

becomes

```
class X {
private:
  int old_var;
  int new_var_1;
  int new_var_2;
  class D {int new_var_3;
           int new_var_4;};
  D * d;
public:
  X();
  X(const X & x);
  X & operator= (const X & x);
  ~X();
  ...
};


X::X() {d = new D();}
X::X(const X & x) {...; d = new D(*x.d);}
X & X::operator= (const X & x) {...}
X::~X() {delete d;}
```

But accessing the data members is now cumbersome, since we must always use `d->new_var_3` instead of just `new_var_3`. There is also a slight performance hit due to the extra layer of indirection.

With a macro system this solution can easily be automated by writing a macro to do the same thing. For example we could write a macro to recognize a `fix_size` flag to a class as so:

```
class X
  : fix_size(sizeof(long)*4)
{
private:
  int old_var;
public:
  ...
};
```

Then, no end-user visible tricks are needed to add a new member as the size of the object will not change. Thus binary compatibility is maintained. Furthermore, since macros handle the low-level details and not the programmer, this solution will be portable across different architectures; the user does not have to know the exact size the types involved.

If a programmer tries to use more space than is preallocated, a compile-time error will be emitted. If the programmer wishes to allow additional private data members an additional flag can be specified:

```
class X
  : fix_size(sizeof(long)*4), allow_expansion
{
private:
  int old_var;
  int new_var_1;
  int new_var_2;
  int new_var_3;
  int new_var_4;
public:
  ...
};
```

and the macro responsible for implementing this feature will allocate additional space if necessary. In this case the implementation will look a lot like the C++ example just given.

3.2.2 Storing the Private Data in a Separate Object

Another solution is simply to keep all the private data members in a separate object, for example:

```
class X {
private:
  class D {
    int var_1;
    int var_2;
    ...
  };
  Data * d;
public:
  X() {d = new D();}
  X(const X & x) {d = new D(*x.d);}
  X & operator= (const X & x) {...}
  ~X() {delete d;}
};
```

This solution is simpler than reserving space ahead of time since it does not require foresight in how large the object may be now, and in the future, and it also does not depend on the size of the types. This solution, known as the *pimpl idiom*, is in fact a very common solution used in practice. The only downside is that there is additional overhead involved in the creating, copying, and deleting of X, and that all accesses to private data must be done through. Based on our own tests the overall slowdown from using this idiom is anywhere between a factor of 1.0 and 1.8. In practice the slowdown is likely to be closer to 1.0 than 1.8. Appendix A gives more details on the tests performed.

This solution can easily be implemented via macros using a similar syntax as before:

```
class X
  : fix_size, allow_expansion
{
  int var_1;
  int var_2;
}
```

Since no size is given to `fix_size`, it will be assumed that only enough space should be allocated to maintain a pointer to an additional object which will store all the private data members.

3.2.3 Avoiding Direct Allocation

As previously described, the problem with changing the size of the object is that the information is needed if the object is directly allocated. This problem can be avoided

by disallowing the object to be directly allocated using the standard C++ trick of making constructors, assignment, and destructors private, and instead provide methods to create, copy, and destroy the function:

```
class X {
private:
  X();
  X(const X &);
  void operator=(const X &);
  ~X();
public:
  X * clone();
  static X * allocate();
  static X * destroy();
};
```

This strategy means that the object cannot be directly allocated on the stack or embedded in other objects. However, it also means that the object cannot be allocated using C++ builtin `new` and `delete`, since `new` and `delete` cannot be overloaded on a per-class bases.

If `new` and `delete` are implemented via macros, then they can easily be modified to use a different approach for a particular object.

3.2.4 Why Not a Fixed Set of Language Extensions?

Reserving space ahead of time is a good solution when performance really matters. However since it requires planning ahead and depends on knowing the exact size of types it is not a very attractive option. Storing all the private data in a separate object is easier to implement, but it does have a small performance overhead which some may find unacceptable. Finally, preventing direct allocation is an undesirable alternative to users of the library.

Since none of these solutions is perfect, none of them are good candidates for language extensions. However, in a system where higher-level objects are implemented using macros the programmer is free to extend these macros to support whichever solution is best suited to the problem.

3.3 Adding New Virtual Methods

A C++ ABI can also change by adding new virtual methods since that changes the size of the vtable. This is a problem because the vtable is often included in the object file instead of being created dynamically at load time.

The C++ solution is similar to the one for adding new data members, except that space is reserved by using dummy methods. For example,

```
class X {
public:
  virtual old_method();
 private:
  virtual void _dummy_1();
  virtual void _dummy_2();
  virtual void _dummy_3();
};
```

reserves enough space for three new methods. Then to add a new method simply replace one of them with the real method:

```
class X {
public:
  virtual old_method();
  virtual new_method();
private:
 virtual void _dummy_2();
 virtual void _dummy_3();
};
```

In a system where macros will be used to implement inheritance, these macros can easily be expanded to reserve space ahead of time for additional methods by recognizing syntax similar to:

```
class X
  : vtable_slots(4)
{
   ...
}
```

which will create a virtual table with four slots in it.

However, unlike the case of adding private data members, running out of slots is a serious problem as there is no way to simply add a pointer to another object to add more

virtual functions, as we could before. There are still ways to add functionality to the class; however, it can not be through adding more virtual methods.

Thus, the idea of fixing the size of the vtable is not an attractive solution since it requires the programmer of the library to have foresight into how many virtual functions they will ever need for this object. Therefore, support for this strategy is not something that is likely to get added to as a language extension. But as before, in some situations, it may still be a viable option.

3.4 Reordering

Another way to break ABI compatibility in C++ is to reorder the methods or the instance variables since that will change the offsets in the vtable or the object's instance, respectively. For example changing:

```
class X {
public:
  virtual a();
  virtual b();
};
```

to

```
class X {
public:
  virtual b();
  virtual a();
};
```

will change the offset of a and b.

There is no real solution to this in C++ other than to just be aware of this fact and not do it. However there are several ways this problem can be solved by modifying how inheritance is implemented:

1. One idea is to use something like:

```
class X
  : freeze_virtual_table
{
  ...
}
```

which will in a separate interface file store the offset of each virtual function. Once a function is defined its offset will never change.

2. Another idea is to put the virtual methods into groups, something like:

```
class X
{
public (group 1):
  virtual a();
  virtual b();
public (group 2):
  virtual c();
  virtual d();
};
```

   then sort each of the groups alphabetically. When new methods are added put them into another group.

3. Finally, a Java-like approach can be used where the offsets are determined when the class is first used. However, this will involve a completely different ABI from the one generally used in C++.

The problem of reordering instance variables can be solved using similar techniques, except that changes will affect class instances and not the vtable.

None of these solutions are particularly attractive; thus, any one of them is unlikely to be implemented as an extension to C++. However, just because no single solution is attractive does not mean that they are not useful. With a macro system in which classes are implemented via macros, the programmer is free to chose which solution, if any, is best for the particular situation.

3.5 Removing Members

Removing methods and instance variables breaks ABI compatibility in C++ since it changes the order the vtable or the object's instance, respectfully.

The only way to solve these problems in C++ is to avoid removing the method or instance variables by instead replacing it with a dummy member. This way the layout is preserved. The unused slot can then later be replaced with a new member in order to save, or it can simply be left unused.

A macro system can help automate this method by freezing the layout in a very similar fashion as discussed in Section 3.4. When this is done the macro will automatically insert a dummy member in place of the removed member. Later on the unused slot can be replaced with a new member.

3.6 Migrating Method Upwards

Migrating a method upwards in the class hierarchy breaks binary compatibility in C++. Solving this problem in C++ is not really possible, but a macro system can help as long as space is reserved ahead of time so that it is is possible to add new virtual methods to the class (as discussed in section 3.3) and single inheritance is used. The idea is as follows: instead of actually removing the method from the old class, the vtable for the old class is adjusted to simply point to the method in the new class. This technique will not work when multiple inheritance is used, since the pointer to the class instance may need to be adjusted; in that case, a proxy method can be created that will just call the new method.

3.7 Adding Parameters

Adding parameters to a function in C++ will break binary compatibility because it changes the name of the symbol used to represent the function. The name changes because, to support overloading, the types of the parameters are encoded as part of the name. However, since C++ allows for overloading it is possible to define a new function with the added parameter. The old function can then call the new one. Macros can automate this technique.

3.8 Other Difficult Transformations

Unfortunately, some program transformations are difficult if not impossible to support within the constraints of an existing ABI. Such transformations include inserting a new class and migrating a parent downwards in the class hierarchy. The transformations are difficult because the parent class is directly embedded in the child class, for both instances of the classes and the vtable. The only real way to support these transformations is to define a better API.

3.9 A Better ABI

All of the solutions in the previous sections work within an existing ABI; thus none of the solutions were ideal. However, the problems can all be solved by defining a new ABI that takes these transformations into account. The new ABI can support transformations that are difficult if not impossible to solve within the existing C++ ABI such as those mentioned in section 3.8.

With a system where a large part of what affects the ABI is written using macros, it is possible to write a new ABI from scratch in order to minimize the issue of ABI fragility, for example implementing something similar to the Object Binary Interface [38], or maybe even implementing something close to what Java does.

However, using a new ABI is not always an option. For example, if a library developer wants code to be usable by existing C++ applications, the developer must use the existing C++ ABI. Thus it is necessary to give programmers the tools to work within an existing ABI when necessary, but also give programmers the option of creating a new ABI when appropriate.

3.10 Changing Compilers

Finally, changing compilers also breaks binary compatibility, since ABIs differ between compilers and sometimes between different versions of the same compiler. Thus, when using C++ libraries, not only is the specific version of the library important, but so is the compiler used to compile it. Unfortunately, there is no good solution to this problem in C++, other than always using a compatible compiler when compiling the library. The only way to support a different, incompatible, compiler is to avoid directly using the C++ ABI altogether. A typical work-around is to create a C API on top of the internal C++ API, and then only export the C API. This technique effectively defines a program-specific ABI that the library developer has complete control over. It may seem silly for a C++ program to have to use a C API to use another C++ library, but currently there is no other way around the problem.

However, if classes are implemented via macros, then the programmer has control of how classes are implemented, and thus has control over which ABI is used. In fact, a programmer can use classes with different ABIs within the same program. For example,

the ABI used can be specified as part of the class declaration. For using existing code, the ABI can be specified on a per header-file basis. Via the right macro hooks, additional parts of the the ABI, such as name mangling, can be brought under control of the programmer.

CHAPTER 4

ZL OVERVIEW

ZL is a C++-compatible language that solves ABI compatibility problems by giving the programmer as much control as possible. ZL provides a C-like core and enough of C++ to let the type-checker and compiler do its job without committing to key parts of the ABI such as class layout. The rest is defined using a sophisticated macro system.

The ZL library provides a default implementation of language constructs such as classes. The implementation can be overridden or extended by defining new macros in a source file or by importing a macro library. Macros, including those that define the behavior of a language construct, are scoped and can be shadowed. This means it is possible to use two different class ABIs by loading one class library and defining some classes, then loading another library and defining some more classes. A more convenient solution is to add some syntax for selecting the ABI for a class, which ZL also supports.

This chapter gives an overview of ZL. Chapters 7 and 8 will give a more detailed description of ZL and how it is implemented.

4.1 ZL Primitives

Most of the class implementation in ZL is left to macros, but since classes are an integral part of the C++ type system, ZL still needs to have some notion of what a class is. *User types* are ZL's minimal notion of classes. A user type has two parts: a type, generally a `struct`, to hold the data for the class instance, and a collection of symbols for manipulating the data.

The collection of symbols is a *module*. For example,

```
module M { int x;
           int foo(); }
```

defines a module with two symbols. Module symbols are used by either importing them into the current namespace, or by using the special syntax `M::x`, which accesses the `x` variable in the above module.

A user type is created by using the `user_type` primitive, which serves as the module associated with the user type. A type for the instance data is specified using `associate_type`.

As an example, the class[1]

```
class C { int i;
          int f(int j) {return i + j;} };
```

roughly expands to:

```
user_type C {
  struct Data {int i;};
  associate_type struct Data;
  macro i (:this ths = this) {...}
  macro f(j, :this ths = this) {f`internal(this, j);}
  int f`internal(...) {...}
}
```

which creates a user type `C` to represent a class `C`; the structural type `Data` is used for the underlying storage. The macro `i` implements the `i` field, while the `f` macro implements the `f` method by calling the function `f`internal` with `ths` as the first parameter. The next section will explain the syntax of the macros and Section 7.1 will give the full expansion of class `C` and a more complete picture of how classes are implemented.

ZL also supports syntax for creating macros, of which there are two kinds: *pattern-based macros* that simply rearrange syntax, and *procedural macros* that are functions that perform more complex manipulation of syntax or take action based on the input, as is necessary to implement classes.

4.2 Macros

The simplest form of a macro is a *pattern-based macro*, which is simply a transformation of one piece of syntax to another. For example, consider an `or` macro that behaves

[1] For simplicity, we leave off access control declarations and assume all members are public in this dissertation when the distinction is unimportant.

like C's `||` operator, but instead of returning true or false, returns the first nonzero value. Thus, `or(0.0, 6.8)` returns `6.8`. To define it, one uses ZL's `macro` form, which declares a pattern-based macro:

```
macro or(x, y) { ({typeof(x) t = x; t ? t : y;}); }
```

In ZL, as in GCC, the `({...})` is a statement expression whose value is the result of the last expression, and `typeof(x)` gets the type of a variable. Like Scheme macros [24], ZL macros are hygienic, which means that they respect lexical scope. For example, the `t` used in `or(0.0, t)` and the `t` introduced by the `or` macro remain separate, even though they have the same symbol name.

The `or` macro above has two *positional* parameters. Macros can also have *keyword* parameters and *default values*. For example:

```
macro sort(list, :compar = strcmp) {...}
```

defines the macro `sort`, which takes the keyword argument `compar`, with a default value of `strcmp`. A call to sort will look something like `sort(list, :compar = mycmp)`.

4.3 Parsing and Expanding

The macros shown so far are pattern-based macros. Writing more sophisticated procedural macros, such as those required to implement classes, requires some knowledge of parsing and macro expansion in ZL. This section gives the necessary background material, while the next section details how to write such macros.

To deal with C's idiosyncratic syntax while also allowing the syntax to be extensible, ZL does not parse a program in a single pass. Instead, it uses an iterative-deepening approach to parsing. The program is first separated into a list of partly parsed declarations by a Packrat [34, 35] parser that effectively groups tokens at the level of declarations, statements, grouping curly braces, and parentheses. Each declaration is then parsed. As it is being parsed and macros are expanded, subparts, such as code between grouping characters, are further separated.

ZL's iterative-deepening strategy is needed because ZL does not initially know how to parse any part of the syntax involved with a macro. When ZL encounters something that looks like a function call, such as `f(x + 2, y)`, it does not know if it is a true function

call or a macro use. If it is a macro use, the arguments could be expressions, statements, or arbitrary syntax fragments, depending on the context in which they appear in the expansion. Similarly, ZL cannot directly parse the body of a macro declaration, as it does not know the context in which the macro will ultimately be used.

More precisely, the ZL parsing process involves three intertwined phases. In the first phase *raw text*, such as `(x+2)`, is parsed. Raw text is converted into an intermediate form known as a *syntax object*, which can still have raw-text components. (Throughout this paper we show syntax objects as S-expressions, such as `("()" "x+2")`.) In the second phase, the syntax object is expanded as necessary and transformed into other syntax objects by expanding macros until a fixed point is reached. In the third phase, the fully expanded syntax object is compiled into an *AST*.

Figure 4.1 demonstrates ZL's parsing and expansion process. The top box contains a simple program as raw text, which is first parsed. The result is a *syntax list* (internally represented as a `@`) of `stmt`'s where each `stmt` is essentially a list of tokens, as shown in the second box. Each statement is then expanded and compiled in turn, and is added to the top-level environment (which can be thought of as an AST node). The third box in the figure shows how this is done, which requires recursive parsing and expansion. The first `stmt` is compiled into the `fun f`, while the body of the function is left unparsed. Next, `fun` is compiled into an AST (shown as a rounded rectangle). During the compilation, the body is expanded. Since it is raw text, this process involves parsing it further, which results in a `block`. Parsing the block involves expanding and compiling the subparts. Eventually, all of the subparts are expanded and compiled, and the fully parsed AST is added to the top-level environment. This process is repeated for the function `main`, after which the program is fully compiled.

4.4 Procedural Macros

Some macros must take action based on the input. One example is the built-in class macro. Another example is a macro that fixes the size of the class, since the amount of padding it needs to add depends on the numeric value of the size passed in. For these situations, ZL provides *procedural macros*, which are functions that transform syntax

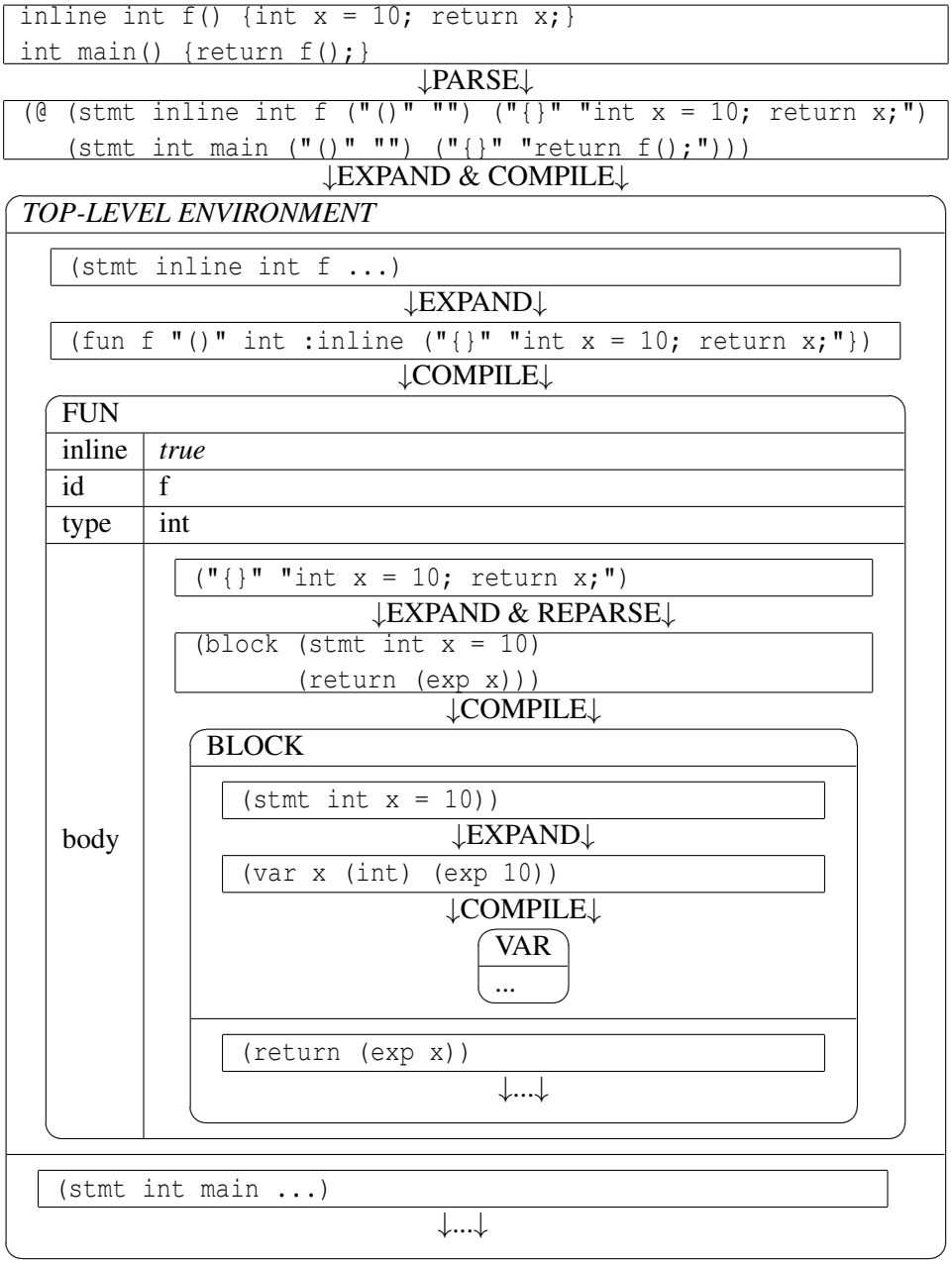

Figure 4.1. How ZL compiles a simple program. The body of `f` is reparsed and expanded as it is being compiled.

objects.

Figure 4.2 demonstrates the essential parts of any procedural macro. The macro is defined as a function that takes a syntax object and environment, and returns a transformed syntax object. Syntax is created using the `syntax` form. The `match` function is used to decompose the input while the `replace` function is used to rebuild the output. Finally, `make_macro` is used to create a macro from a function. More interesting macros use additional API functions to take action based on the input. Figure 4.3 defines the key parts of the macro API, which we describe in the rest of this section.

Syntax is created using the `syntax` and `raw_syntax` forms. The different forms create different types of code fragments. In most cases, the `syntax {...}` form can be used, such as when a code fragment is part of the resulting expansion; the braces will not be in

```
Syntax * or(Syntax * p, Environ *) {
  Match * m = match(NULL, syntax (_, x, y), p);
  return replace(syntax
                   {({typeof(x) t = x; t ? t : y;});},
                 m, new_mark());
}
make_macro or;
```

Figure 4.2. Procedural macro version of `or` macro from Section 4.2.

Types: `UnmarkedSyntax`, `Syntax`, `Match`, and `Mark`

Syntax forms:

```
new_mark() — returns Mark *
syntax (...)|{...}|ID — returns UnmarkedSyntax *
raw_syntax (...) — returns UnmarkedSyntax *
make_macro ID [ID];
```

Callback functions:

```
Match * match(Match * prev, UnmarkedSyntax * pattern, Syntax * with)
Match * match_args(Match *, UnmarkedSyntax * pattern, Syntax * with)
Syntax * match_var(Match *, UnmarkedSyntax * var);
Syntax * replace(UnmarkedSyntax *, Match *, Mark *)
size_t ct_value(Syntax *, Environ *)
```

Figure 4.3. Basic macro API.

the resulting syntax. If an explicit list is needed, for example, when passed to `match` as in Figure 4.2, then the `syntax (...)` form should be used (in which the commas are part of the syntax used to create the list). Neither of these forms create syntax directly, however; for example, `syntax {x + y;}` is first parsed as `("{}" "x + y;")` before eventually becoming `(plus x y)`. When it is necessary to create syntax directly, the `syntax ID` form can be used for simple identifiers. For more complicated fragments the `raw_syntax` form can be used in which the syntax is given in S-expression form.

The `match` function decomposes the input. It matches pattern variables (the second parameter) with the arguments of the macro (the third parameter). If it is successful, it prepends the results to `prev` (the first parameter) and returns the new list. If `prev` is `NULL`, then it is treated as an empty list. In the match pattern a _ can be used to mean "don't care." The match is done from the first part of the syntax object. That is, given `(plus x y)`, the first match is `plus`. Since the first part is generally not relevant, ZL provides `match_args`, which is like `match` except that the first part is ignored. For example, `match_args` could have been used instead of `match` in Figure 4.2.

The `replace` function is used to rebuild the output. It takes a syntax object (the first parameter, and generally created with `syntax`), replaces the pattern variables inside it with the values stored in the `Match` object (the second parameter), and returns a new `Syntax` object.

The final argument to `replace` is the *mark*, which is used to implement hygiene. A mark captures the lexical context at the point where it is created. Syntax objects created with `syntax` do not have any lexical information associated with them, and are thus *unmarked* (represented with the type `UnmarkedSyntax`). It is therefore necessary for `replace` to attach lexical information to the syntax object by using the mark created with the `new_mark` primitive (the third parameter to `replace`).

Match variables exist only inside the `Match` object. When it is necessary to access them directly, for example, to get a compile-time value, `match_var` can be used; it returns the variable as a `Syntax` object, or `NULL` if the match variable does not exist. If the compile-time value of a syntax object is needed, `ct_value` can be used, which will expand and parse the syntax object and return the value as an integer.

Once the function for a procedural macro is defined, it must be declared as a macro using `make_macro`.

This section only gives a small part of the macro API. A more detailed description is given in Chapter 7. Some of the more important functions not shown here include functions for controlling the visibility of macros and partly expanding syntax.

4.5 The Class Macro

We have now presented most of the necessary parts that make up the class macro. Sections 4.1 and 4.2 give a representation of the code generated, while Sections 4.3 and 4.4 give a representation of what is necessary to generate that code. The remaining details are given in Chapter 7, which includes more of ZL's macro API. The class macro also uses ZL's support for *syntax macros*, which work with arbitrary syntax, as opposed to *function-call macros*, which only work with syntax that takes the shape of a function call or identifier.

The core class macro is currently around 900 lines of code. The implementation is highly reusable, because it is a class itself that is organized around methods that can be overridden to extend its functionality. The bootstrapping problem of writing methods to implement classes is solved by having a simpler, more compact class system just to implement the class macro.

In addition to overriding individual methods, the `class` syntax object can be declared to expand to a completely different macro. The class macro is defined using the function `parse_class`, which can be called directly so that the new macro can reuse the original implementation.

CHAPTER 5

USING ZL TO MITIGATE ABI PROBLEMS

ZL can be used to mitigate key ABI problems discussed in Chapter 3. This chapter gives the details of how key techniques from that chapter are implemented in ZL (see Table 5.1 for an overview). The next chapter demonstrates how these techniques can be used to mitigate binary compatibility problems through the evolution of a simple spell checker.

5.1 Adding Data Members without Changing Class Size

Adding data members to a class changes the size of the class, which breaks binary compatibility. To avoid this problem we must somehow fix the size of the class.

Table 5.1. ZL's solution for changes that can affect the ABI. ZL can implement all of the techniques discussed in Chapter 3 (and shown in Table 3.1, page 12). However, only a key subset of the techniques discussed are currently implemented. An outline of how ZL can implement the other techniques is given in Section 5.7.

Change	Solution Implemented	Section
add instance variable	Yes	5.1, 5.3
add new method	Yes	5.2, 5.3
reorder methods	-	5.7
reorder instance variables	-	5.7
remove private method	-	5.7
remove private instance variable	-	5.7
migrate method upward in class hierarchy	-	5.7
add parameters	-	5.7
insert new class in class hierarchy	-	5.7
migrate parent downward in class hierarchy	-	5.7
change compilers	Yes	5.5, 5.6

5.1.1 Fixing the Size of a Class

As described in Section 3, one common technique to fix the size of the class is to add dummy data members as placeholders to allow for future expansion. Using the ZL macro system, it is possible to automate this solution, as shown in Figure 5.1. To support this extension the ZL grammar has been enhanced to support specifying the size. The syntax for the new class form is:

```
class C : fix_size(20) {...};
```

which allows a macro to fix the size of the class `C` to 20 bytes.

The macro in Figure 5.1 redefines the built in `class` macro. It works by parsing the class declaration and taking its size. If the size is smaller than the required size, an array of characters is added to the end of the class to make it the required size.

The details are as follows. Lines 2–7 decompose the class syntax object to extract the relevant parts of the class declaration. A `@` by itself in a pattern makes the parts afterward optional. The `pattern` form matches the subparts of a syntax object; the first part of the object (the `{...}` in this case) is a literal[1] to match against, and the other parts of the object are pattern variables. A `@` followed by an identifier matches any remaining parameters and stores them in a syntax list; thus, `body` contains a list of the declarations for the class. Finally, `:(fix_size fix_size)` matches an optional keyword argument; the first `fix_size` is the keyword to match, and the second `fix_size` is a pattern variable to hold the matched argument.

If the class does not have a body (i.e., a forward declaration) or a declared `fix_size`, then the class is passed on to the original class macro in line 9. Line 11 compiles the `fix_size` syntax object to get an integer value.

Lines 13–22 involve finding the original size of the class. Due to alignment issues the `sizeof` operator cannot be used, since a class such as "`class D {int x; char c;}`" has a packed size of 5 on most 32 bit architectures, but `sizeof(D)` will return 8. Thus, to get the packed size, a dummy member is added to the class. For example, the class `D` will become "`class D {int x; char c; char dummy;}`" and then the offset of the dummy

[1] ZL matches literals symbolically (i.e., not based on lexical context). Matching sensitive to lexical context is future work. (See 11.5.2)

```
1  Syntax * parse_myclass(Syntax * p, Environ * env) {
2    Mark * mark = new_mark();
3    Match * m = match_args
4        (0, raw_syntax(name @ (pattern ({...} @body))
5                       :(fix_size fix_size) @rest), p);
6    Syntax * body = match_var(m, syntax body);
7    Syntax * fix_size_s = match_var(m, syntax fix_size);
8
9    if (!body || !fix_size_s) return parse_class(p, env);
10
11   size_t fix_size = ct_value(fix_size_s, env);
12
13   m = match(m, syntax dummy_decl,
14             replace(syntax {char dummy;}, NULL, mark));
15   Syntax * tmp_class = replace(raw_syntax
16       (class name ({...} @body dummy_decl) @rest),
17                                m, mark);
18   Environ * lenv = temp_environ(env);
19   pre_parse(tmp_class, lenv);
20   size_t size = ct_value
21       (replace(syntax(offsetof(name, dummy)), m, mark),
22        lenv);
23
24   if (size == fix_size)
25     return replace(raw_syntax
26                      (class name ({...} @body) @rest),
27                    m, mark);
28   else if (size < fix_size) {
29     char buf[32];
30     snprintf(buf, 32, "{char d[%u];}", fix_size - size);
31     m = match(m, syntax buf,
32             replace(string_to_syntax(buf), NULL, mark));
33     return replace(raw_syntax
34                      (class name ({...} @body buf) @rest),
35                    m, mark);
36   } else
37     return error(p,"Size of class larger than fix_size");
38 }
39 make_syntax_macro class parse_myclass;
```

Figure 5.1. Macro to fix the size of a class. All . . . in this figure are literal.

member with respect to the class `D` is taken. This new class is created in lines 13–17. Here, the `@` before the identifier in the replacement template splices in the values of the syntax list.

To take the offset of the dummy member of the temporary class, it is necessary to parse the class and get it into an environment. However, we do not want to affect the outside environment with the temporary class. Thus, a new temporary environment is created in line 18 using the `temp_environ` macro API function. Line 19 then parses the new class and adds it to the temporary environment. The `pre_parse` API function partly expands the passed-in syntax object and then parses just enough of the result to get basic information about symbols.

With the temporary class now parsed, lines 20–22 get the size of the class using the `offsetof` primitive.

Lines 24–37 then act based on the size of the class. If the size is the same as the desired size, there is nothing to do and the class is reconstructed without the `fix_size` property (lines 24–27). If the class size is smaller than the desired size, then the class is reconstructed with an array of characters at the end to get the desired size (lines 28–35). (The `string_to_syntax` API function simply converts a string to a syntax object.) Finally, an error is returned if the class size is larger than the desired size (lines 36–37).

The last line declares the function `parse_myclass` as a syntax macro for the `class` syntax form.

5.1.2 Allowing Expansion

The example in Figure 5.1 demonstrates one technique for preserving binary compatibility when adding new data members. However, this technique requires planning ahead and reserving enough space for all future extensions. If there is not enough space reserved but enough space for a pointer, then the remaining space can be used to point to the rest of the data. For example:

```
class C : fix_size(12) { int x; int y; int i; int j; };
```

could become:

```
class C { int x; int y;
          struct {int i; int j;} * data; }
```

To do this, we modify the macro definition in Figure 5.1 to use the last bit of available space for the overflow pointer instead of returning an error. To a user of the class, the fact that some data members are stored in a separate object is completely transparent. In the above example, if `x` is an instance of class `C`, then data member `i` can be accessed using `x.i`. The full expansion of class `C` is something like:

```
class C { int x; int y;
          class Overflow {
              struct Data { int i; int j; };
              struct Data * ptr;
              Overflow() {ptr = malloc(sizeof(Data));}
              Overflow(const Overflow & o)
                  {ptr = malloc(sizeof(Data));}
              ~Overflow() {free(ptr);} };
          Overflow overflow;
          pseudo_member i int overflow.ptr->i;
          pseudo_member j int overflow.ptr->j; };
```

The key to making this work is the use of `pseudo_member` (which is built into the default class macro) to create pseudo members that behave like normal members for most purposes. This support includes properly calling the constructor and destructor for the member if it has one. Thus, the members in `C::Overflow::Data` will get properly initialized even though `malloc`/`free` is used instead of `new` and `delete`.

In principle, the `fix_size` macro can work without the `pseudo_member` extension, but doing so greatly increases the complexity of `fix_size`, and implementing `pseudo_member` in the class macro was accomplished in around 6 lines of code. In addition, a closely related feature, `alias`, is useful for implementing other features such as anonymous unions. An `alias` is like a `pseudo_member` except that the constructor and destructor for the member are not called.

We chose to implement `pseudo_member` in the default macro class. However, since the class macro is built using its own class system, extending the class macro to support `pseudo_member` is fairly straightforward, and would still be less work than trying to do all the work in the `fix_size` macro.

The enhanced `fix_size` macro can also be used to store all the private data, i.e. the "pimpl idiom," in a separate object by specifying a size of zero, which the `fix_size` macro would recognize as a special case.

5.1.3 Validation

Both previously mentioned techniques have been implemented in ZL as a macro library. All the end user needs to do is include the library, which will replace the class implementation with one that supports fixing the size. We have verified that the size does not change under various scenarios and hence binary comparability will be maintained.

5.2 Fixing the Size of the Virtual Table

Adding new virtual methods can break binary compatibility in essentially the same way as adding data members. Since the macro that implements classes uses another class to implement the vtable, all of the techniques previously discussed can easily be used to fix the size of the vtable. To make this strategy work, the ZL class macro provides a way to specify the implementation of the class used to implement the virtual table.

We have written a macro that uses the technique just described to allowing fixing the vtable size using the special syntax:

```
class X : fix_vtable_size(8) {...}
```

which will fix the vtable size to 8 bytes. We have verified that the macro does indeed fix the size of the vtable and hence maintains that aspect of binary compatibility. We have also written a a more sophisticated macro that, amount other things, allows the size to be implemented in terms of slots, which is discussed in Section 6.6.

5.3 A Better ABI

Adding new data members or methods breaks binary compatibility because the sizes of the class and vtable are needed at compile time. The size of the class is needed when directly allocating an object on the stack, or when inlining one object into another. The first can be avoided by dynamically allocating the class on the heap. However, the second is a problem with most C++ ABIs as a typical C++ ABI defines class layout to be something like:

```
class Parent {...};
class Child { Parent parent; ...};
```

which inlines the parent in the child class. This means adding new data members to the parent class will break binary compatibility for any code that depends on the child class.

We defined a new ABI to avoid this problem. Our new ABI defines class layout to be something like:

```
class Parent {void * child_ptr; ...};
class Child {
  Parent * parent_ptr; void * child_ptr; ...};
```

where the parent class is dynamically allocated when the child class is created, and `child_ptr` is used to downcast. This strategy preserves binary compatibility when new data members are added to the parent. A similar strategy is used for the vtable.

The code to implement the new ABI is under 60 lines of code. It overrides three methods from the core class macro; the method that adds the parent info to the user type was rewritten, and some additional information was added to every user type to include the child pointer.

We verified that the new ABI maintains binary comparability when adding new data members by creating a situation in which adding data members would cause problems with the more traditional ABIs. For example, in the following code:

```
class X {int x;}
class Z : public X {int z;}
```

adding a new data member, say `y`, to `X` will break binary compatibility with programs that use `Z` since the addition will change the offset of `z`. Therefore, accesses to the data member `z` will report an incorrect value. We verified that this was indeed a problem with ZL's default ABI, by setting the value of `z` with object code compiled against the new API (the one with the new `y` data member) but reading the value with object code compiled against the original API (without `y`) and verified that a different value was returned. We then did the same thing with the new ABI and verified that the same value was returned. We did a similar test to verify that adding new virtual methods will not break binary comparability.

For many purposes, this ABI can impose too much overhead. For example, each class must have a pointer to the child to support down casting, and virtual-method dispatch is slower. When binary compatibility is a primary concern, however, this ABI can be a good choice. Furthermore, since ZL can use more than one ABI at a time, a programmer can choose this ABI for just the parts of a program where the benefits in binary compatibility outweigh the costs in performance.

5.4 Matching an Existing ABI

Because classes are just user types to the compiler, it is possible to construct classes to match an existing ABI. This includes specialized ABIs which are really a C implementation of classes (such as done in GNOME [5]) or C wrappers around a C++ API (such as done in Aspell [1]). Doing so provides a more class-like interface to the C API. For example, ZL's macro API is a pure C API for simplicity; however, a more class-like interface is also provided. ZL provides a class-like interface to many of the API types including `Match`, `Syntax`, and `UnmarkedSyntax`. For example, instead of using `match_var(m, syntax x)`, one can use `m->var(syntax x)`. This is done by creating a user type `Match` that looks something like:

```
user_type Match { associate_type struct Match;
                  macro var(str, :this ths)
                      { match_var(ths, str);} };
```

5.5 Matching GCC's ABI

Just as it is possible to match a C ABI, it is possible to match other compilers' ABIs. It is even possible to use classes with different ABIs in the same program with some restrictions, which depend on fundamental incompatibilities between different ABIs. For example, while it is possible to mix classes with different ABIs through composition, doing so via inheritance is unlikely to work. This is due to differences in how inheritance is implemented, and in particular, how the vtable is laid out.

To demonstrate that ZL is complete enough to match another compiler's ABI we have matched the GCC ABI. The vtable layout turned out to be compatible with ZL's default ABI. However, there were still some key differences between ZL's default ABI and GCC. The most significant one is that each class has multiple implementations of each constructor and destructor. In particular there is an allocating constructor that calls `new` and then constructs the object, the constructor that is called by derived classes, and the normal constructor. In a similar fashion there are multiple destructors. If the destructor is virtual than there are also multiple destructors in the vtable, hence affecting vtable layout. In addition to class layout, the mangling scheme used by GCC is different from that of ZL.

The code to implement the GCC ABI consists of around 150 lines of code to extend the class macro and around 300 lines of code to implement the alternative mangling scheme. A demonstration that we indeed matched the GCC ABI is given throughout the next chapter.

5.6 Matching Another ABI

In addition to the ABIs we have already implemented, we implemented one additional ABI. We defined a new ABI by building on the existing class macro to pass the `this` parameter as a global variable. This implementation simulates passing the `this` parameter in a register, as the Microsoft C++ ABI does, as opposed to passing it as the first parameter, as GCC does. We then used both ABIs in the same program, and even embedded classes with one ABI in another via composition. The code to implement the new ABI was under 45 lines. The only methods from the core class macro that needed to be overridden were the ones involved with constructing and calling member functions—three in all.

5.7 Other ABI Problems

This chapter illustrated how ZL enabled solutions to key ABI problems outlined in Chapter 3. There are no fundamental limitations to solving the other ABI problems outlined in that chapter. Regarding techniques for maintaining binary compatibility, it is just a matter of writing the macros to implement the additional techniques. The main difficulty in the unimplemented techniques is the bookkeeping to keep track of previous states in the ABI. For example to avoid breaking binary compatibility when reordering methods it is necessary to keep track of the old layout somehow. While possible with ZL macros—as they can perform I/O—there is still some issues to work out before they can be made reliable.

CHAPTER 6

THE CASE OF A SIMPLE SPELL CHECKER

In this chapter we use the techniques of the previous chapter to mitigate binary compatibility problems through the evolution of a simple spell checker (which we refer to as Simple Spell). In addition, we demonstrate ABI compatibly with GCC.

6.1 Simple Spell

Simple Spell is a spell checker that provides basic spell checker functionally. It can check that a word is in a dictionary and handles case in a intelligent fashion; for example, if a first letter is upper case, Simple Spell will first try to match the word in a case-sensitive fashion, and if that fails, it looks for an all lower-case version of the word; thus, it will reject "Mcdonald" as the correct spelling is "McDonald," but still accept "Color" and "Dog." If a word is not found in the dictionary then Simple Spell will offer a list of words which are within one edit-distance of the misspelled word. For example, it will suggest "the" when given "teh" or "color" when given "colr." If the misspelled word is indeed the correct spelling, Simple Spell can remember the word to avoiding flagging it again via a session dictionary.

In addition to offering basic spell checker services such as checking if a word is the dictionary, Simple Spell also provides an API for checking documents. The document checker provides the ability to skip over parts of the document that should not be spell checked, such as URL's, via a plugable filter interface that selectively blanks out part of the document. A simple URL filter is provided.

6.2 The Spell Checker API

With concern to binary compatibility there are two API's of interest; there is the API for applications simply wishing to use the spell checker and there is the extension API for

those wishing to extend the functionally of the spell checker. So that Simple Spell can be used by more than just ZL, we will use GCC's ABI for the application ABI. The extension ABI will be in ZL's own ABI so that we have more flexibility in the techniques used to mitigate ABI compatibility problems. Nevertheless, we will still be able to make use of extensions compiled with GCC through the use of a bridge class which will be discussed in a latter section.

6.2.1 The Application API

The most important class in Simple Spell is the `Speller` class, which is responsible for checking that a word is correctly spelled—and when it is not, coming up with a list of suggestions. The definition of the `Speller` class is defined in `speller.hpp` as shown in Figure 6.1.

The basic usage of Simple Spell is to create a new instance of the `Speller` class and then initialize it via the `init` method by giving it a language and dictionary class. The API of those two classes is of no interest to the application writer, instead new instances are created via the `new_lang` and `new_master_dict` factory functions, which are shown in Figure 6.2. Once a new instance of the `Speller` class is created, the `check` method is used to check if a word is the correct spelling. If the word is not the correct spelling the `suggest` method can be used to come up with a list of possible replacements, or if the word is indeed correct, the `add_to_session` method can be used to ignore the word for the rest of the session.

The `Suggestions` struct is used for iterating through the suggestion results. It is a simple wrapper class and as such the implementation details are completely exposed via the header file for the sake of efficiency. The `SugsData` struct is an internal class used by ZL, but its existence must be exposed in the header file since it is a data member of the Speller class.

The preprocessor macros `GCC_ABI_BEGIN` and `GCC_ABI_END` ensure that the GCC ABI is used when compiled with ZL. They are defined in the `config.hpp` header file as such:

```
...
#include "config.hpp"
...
GCC_ABI_BEGIN

struct Suggestions {
  ...
  const_iterator begin() const {return begin_;}
  const_iterator end() const {return end_;}
  const char * operator[](unsigned n) const {return begin_[n];}
  unsigned size() const {return end_ - begin_;}
};

struct SugsData;

class Speller {
  ...
  SugsData * sugs_data;
  ...
public:
  Speller();
  void init(Language * lang, Dictionary * main);
  bool check(const char *);
  void add_to_session(const char *);
  Suggestions * suggest(const char *);
  ~Speller();
private:
  Speller(const Speller &); // no copy
};

GCC_ABI_END
```

Figure 6.1. The `speller.hpp` header file providing the core functionally of Simple Spell.

```
Language * new_lang(const char * name);

Dictionary * new_master_dict(Language *, const char * fn);
WritableDict * new_session_dict(Language *);
```

Figure 6.2. Other parts of the core simple spell API defined in other header files.

```
#ifdef __zl
#  define GCC_ABI extern "C++" : "GCC"
#  define GCC_ABI_BEGIN extern "C++" : "GCC" {
#  define GCC_ABI_END }
#else
#  define GCC_ABI
#  define GCC_ABI_BEGIN
#  define GCC_ABI_END
#endif
```

where `__zl` is defined by the `zlc` compiler when prepossessing ZL code and "`extern "C++" : "GCC"`" selects a macro-pluggable ABI implementation (details in 7.5.3).

The other important part of the spell checker API is the document checker interface, which is shown in Figure 6.3. The `Session` class provides basic document checker support and the `SessionWFilters` class extends it with basic filter support.

A document is checked one line at a time by passing in a line with the `new_line` method. The `next_misspelling` method is then used to advance to the next misspelled word on the current line, assuming there is one, otherwise it returns false. When there is a misspelled word `misspelled_word` returns the word, and `misspelled_offset` and `misspelled_len` can be used to find the word in the current line. If the misspelled word was replaced with another, supposedly correct word, the `replace` method needs to be used to inform the document checker of the correct spelling. When this method is used the checker will recheck the word to make sure it is correct before advancing on.

The extended document checker interface `SessionWFilters` functionally is identical to `Session` except that before checking the document one or more filters needs to be added using the `add` method. The function `new_url_filter` returns a new instance of the URL filter. The details of the `Filter` class are part of the extension API.

6.2.2 The Extension API

Simple Spell supports the ability to provide custom filters by extending the `Filter` class defined in Figure 6.4. New filters simply define the `filter` method, which blanks out any part of the line that should not be spell checked. New filters can then be added via the `add` method of the already shown `SessionWFilters` class (Figure 6.3).

```
class Session {
protected:
  char * word;
  unsigned misspelled_start;
  unsigned misspelled_stop;
  ...
public:
  Session(Speller * sp);
  virtual void new_line(const char *);
  virtual bool next_misspelling();
  virtual void replace(const char * new_word);
  const char * misspelled_word() {return word;}
  unsigned misspelled_offset() {return misspelled_start;}
  unsigned misspelled_len() {return misspelled_stop - misspelled_start;}
  virtual ~Session();
};

class SessionWFilters : public Session {
  ...
public:
  SessionWFilters(Speller * sp);
  virtual SessionWFilters & add(Filter *);
  virtual void new_line(const char *);
  virtual ~SessionWFilters();
};

Filter * new_url_filter();
```

Figure 6.3. Simple Spell document checker API. All parts of this API use the GCC ABI.

```
class Filter {
public:
  Filter() : next() {}
  virtual void filter(char * line) = 0;
  virtual ~Filter();
  Filter * next;
};
```

Figure 6.4. Simple Spell extension API.

6.3 A Simple Application and Binary Compatibility

To demonstrate the functionally of Simple Spell, and that we matched another compilers ABI, we wrote a simple application that uses the Simple Spell library. When given a file name (via the command line) the application checks the current document. Otherwise, it enters a simple demonstration mode that accepts one word per line and reports it as either correct or incorrect and then offers a list of suggestions.

The interface for checking a document is simple but functional. It checks the provided text file for spelling errors and when one is found, prints out the line with the misspelled word highlighted, offers a list of suggestions, and then prompts the user for what to do next. For example:

```
*Teh* dog swm up the stream.
1) The  2) Tea  3) Ted  4) Tee  5) Tel  6) Ten  7) Tet  8) TeX  9) Tech
i) Ignore  I) Ignore all  r) Replace  a) Abort
```

The user can then either accept one of the suggestions, ignore the word this time, ignore the word for the rest of the document (i.e., add it to the session dictionary), offer a replacement, or abort. When done checking the document, a new file is written out that has the same name as the original file but with the `.new` extension added.

We have compiled this simple application with GCC and linked it with a version of Simple Spell compiled with ZL, thus demonstrating that we have indeed matched GCC ABIs with ZL. In addition we have compiled the application with ZL and linked it with a version of Simple Spell compiled with GCC, thus further demonstrating ABI compatibility.

6.4 Adding a Filter, Compiled with GCC

Due to the choice of using ZL's ABI for the `Filter` class, the Simple Spell library, when compiled with ZL, can not make direct use of a `Filter` class that is compiled with GCC. However, we can rectify this situation through the use of a simple bridge class.

6.4.1 The Bridge Class

The bridge class is shown in Figure 6.5. The `Filter` class with ZL's ABI is included in the file `filter.hpp`, while the `Filter` class in GCC's ABI in wrapped in a module so that we can refer to both at the same time. Normally, the module name will appear as part of

```
#include "filter.hpp"

module GCC :asm_hidden {
  GCC_ABI
  class Filter {
  public:
    Filter() : next() {}
    virtual void filter(char * line) = 0;
    virtual ~Filter();
    Filter * next;
  };
}

class FilterBridge : public Filter {
  GCC::Filter * GCC_filter;
public:
  FilterBridge(GCC::Filter * f) : GCC_filter(f) {}
  void filter(char * line) {GCC_filter->filter(line);}
  ~FilterBridge() {delete GCC_filter;}
};

extern "C"
Filter * filter_bridge (void * filter) {
  return new FilterBridge(reinterpret_cast<GCC::Filter *>(filter));
}
```

Figure 6.5. A bridge class to allow using filters compiled with GCC.

the mangled name of symbols defined within it; however, this is clearly not what we want in this case. Thus, the `:asm_hidden` flag is used to make the module invisible as far as external names go. Normally, this will cause name conflicts, but since a different mangling scheme is used for the ZL and GCC ABIs, there is no conflict.

The actual bridge class is fairly simple and should be self explanatory. It implements ZL's `Filter` interface by simply forwarding the `filter` method to GCC `Filter` class.

Directly including header files for any parts involved with the GCC ABI is extremely problematic since there will now be two filter classes, one of which is meant to be in ZL's ABI and others GCC's. A module is not the same thing as a C++ namespace; for example, the following will not work:

```
module GCC {class Filter {...};}
module GCC {class EmailFilter : public Filter {...};}
```

as the second module will shadow the first rather than extending in. Thus, wrapping the header files in the module will not work. When ZL implements C++ namespaces this might be made to work, but for now we simply avoid the need by not referring to the GCC `Filter` class in the parameter for the `filter_bridge` factory function, and instead cast the void pointer to the correct type.

6.4.2 Adding The Email Filter

With this bridge class now written we make use of it to add a filter that is compiled with GCC to our application. The new filter, the email filter, is a simple filter that skips quoted lines. For example given:

```
> This line will be skippd.
This line will be checkd.
```

the word "skippd" will not be checked but the word "checkd" will.

We avoid including the actual definition of the class itself and instead only include the declaration of the factory function:

```
extern "C" void * new_email_filter();
```

We then make use of the email filter by passing in the pointer returned by `new_email_filter` into `filter_bridge` to create a new `Filter` instance using ZL's ABI.

6.4.3 Automating the Creation of the Bridge Class

The filter class is fairly simple with only one real method. The creation of the bridge class for more complicated methods would be a lot more tedious. In addition, there is the burden of keeping the bridge class up-to-date as methods are added or removed from the interface class. Fortunately it is fairly easy to automate the creation of the bridge class with a procedural macro.

Figure 6.6 shows the essential part of the bridge class. In order to avoid having to include the definition of the class in the macro call we extract the original syntax object from the class definition by using `get_symbol_prop` to extract the `syntax_obj` property from the module used to implement the class. The `syntax_obj` property is one of many properties added by the class macro.

Once we have the syntax object for the class we extract the virtual method definition and create the necessary bridge code. We then return the code to define the bridge class. The symbols `OtherAbi` and `Bridge` are lexically scoped and thus we do not need to worry about conflicts with other bridge classes.

With this macro now written we replace the code in Figure 6.5 with

```
mk_bridge(Filter, filter_bridge, "GCC");
```

in the Simple Spell library.

6.5 Adding Support for a Personal Dictionary

No spelling dictionary can include every possible valid word; thus a key feature of almost any spell checker is the ability to maintain a personal dictionary. We would like to be able to add this feature to Simple Spell without breaking binary compatability. Unfortunately, since we allow direct allocation of the `Speller` class (by exposing the class definition, private data members and all) we cannot easily extend the `Speller` class without breaking binary compatability. For one thing, we cannot add private data members as that will change the size of the class instance.

Fortunately, all is not lost as we can still extend the the `Speller` class, we just need to be careful not to change of the size of the class. Doing so using traditional C++ can be very tedious and error prone. However, assuming we are willing to require that the library

```
Syntax * parse_mk_bridge(Syntax * p, Environ * env) {
  Mark * mark = new_mark();
  Match * m = match_args(0, syntax (class_n, fun_n, abi_name), p);
  Syntax * class_syn = get_symbol_prop(m->var(syntax class_n),
                                       syntax syntax_obj, env);
  m = match_args(m, raw_syntax (_ @ (pattern ({...} @body)) @_), class_syn);

  SyntaxList * bridges = new_syntax_list();
  SyntaxEnum * itr = partly_expand_list(m->varl(syntax body), FieldPos, env);
  Syntax * member;
  while ((member = itr->next)) {
    // if memeber is a virtual method, create a
    // forwarding method and append it to bridges list
  }

  UnmarkedSyntax * res = syntax {
    module OtherAbi :asm_hidden {
      extern "C++" : abi_name $1;
    }
    class Bridge : public class_n {
      OtherAbi::class_n * obj;
      Bridge(OtherAbi::class_n * o) : obj(o) {}
      $2;
      ~Bridge() {delete obj;}
    };
    extern "C" class_n * fun_n (void * o) {
      return new Bridge(reinterpret_cast<OtherAbi::class_n *>(o));
    }
  };
  return replace(res, match_local(m, class_syn, bridges, 0), mark);
}

make_macro mk_bridge parse_mk_bridge;
```

Figure 6.6. Part of the `mk_bridge` macro. The real implementation is just under 55 lines of code.

is compiled with ZL, we can use the `fix_size` macro from Section 5.1 to maintain the size of the class and thus maintain binary compatibility. Note that while we require that the *library* be compiled with ZL, we will still be matching the GCC ABI. Thus *applications* that use Simple Spell can still be compiled with either GCC or ZL.

Figure 6.7 shows the part of the header file defining the improved `Speller` class. The header file is designed to be used by both GCC and ZL.

To be able to fix the size of the class, we first must determine what the size of the old class was; thus we create a dummy class, `SpellerOld` for the sole purpose of taking its size. We then use `fix_size` to fix the size of the new `Speller` class. With the class size fixed, we are free to add (or remove) new private data members. We can even reorder existing onces since we do not expose any code (in the form of inline functions) that use the private data members.

We will also naturally need to add some additional methods to the class, but this will not break binary compatability since the `Speller` class does not have a vtable. We can even even overload an existing method, as is done with `init`, without a problem as it is equivalent to adding a new method since the two methods will be mangled differently.

However, since `fix_size` is a ZL construct and this header file is also used by applications compiled with GCC we must also fix the class size in GCC's eyes. To do this we replace the entire class with a character array of the correct size when the header is read by GCC (or other non-ZL compiler). Since the application has no need to access the private data members this is all that is needed to preserve binary compatibility.

It is important to note that while the header file is slightly complicated, it is far simpler than any solution would of been without the aid of the `fix_size` macro. In particular, the changes shown here are the only changes necessary to fix the size of the class. The library code does not need to worry about the fact the some of the private data members are now likely in a separate object, nor does it need to worry about maintaining the object which is likely to be heap allocated.

To test the new functionally and to verify that we still match the GCC ABI, we enhanced our sample application to take advantage of the new personal dictionary. We then compiled the application with both ZL and GCC and linked it with the same library (which now must

```
class SpellerOld {
  // original Speller class private data members
};

class Speller
#ifdef __zl
  : fix_size(sizeof(SpellerOld))
#endif
{
public: // but don't use
#ifdef __zl
  // private data members
  SavableDict * personal;
  // more private data members
#else
  union {
    void * datap; // to make sure the structure is aligned
    char data[sizeof(SpellerOld)];
  };
#endif
public:
  Speller();
  void init(Language * lang, Dictionary * main);
  void init(Language * lang, Dictionary * main, SavableDict * personal); // NEW
  bool check(const char *);
  void add_to_session(const char *);
  void add_to_personal(const char *);  // NEW
  void save_personal();
  Suggestions * suggest(const char *);
  ~Speller();
private:
  Speller(const Speller &); // no copy
};
```

Figure 6.7. Extending the `Speller` class to include support for a personal dictionary.

be compiled with ZL). We also verified that we indeed maintained binary compatability by linking the original application (before the changes in this section) with the new library without recompiling and verified that everything worked.

6.6 A Better ABI to Allow Future Enhancements

Through the use of ZL we were able to extend the `Speller` class without breaking binary compatability. However, the only reason we were able to do this was because we did not need to add any virtual methods to the `Speller` class. If we did, we would not have been able to extend the `Speller` class as adding any virtual methods will change the offset in the vtable for any derived classes. In addition, we would not be able to fix the size of `Speller`'s vtable as we did with the class itself since—unlike with the private data members—the application does directly use the vtable. As such GCC will not be able to use any methods whose pointer is not directly stored in the vtable.

However, with some planning ahead we can create a better initial ABI which allows for easier expansion. For every class whose definition is exposed to the application we will fix the size of the class from the start. We will also fix the size of the vtable to allow for future expansion. As long as the vtable size is larger than the required size we will not create ABI problems for GCC as a separate object will not need to be used. We will also take use this opportunity to hide some unnecessary implementation details from the application.

Figure 6.8 shows the new `Speller` class definition. Since the applicaton has no need to access any of the private data members we chose to use `fix_size` to implement the pimpl idiom. By doing so we also able to avoid a layer of indirection in the orignal API; in the orignal API (Figure 6.1) the `Speller` class contained a pointer to the `SugsData` class to avoid having to expose the `SugsData` definaton in the header file. However, with the pimpl idiom this is unnecessary since only the `Speller` class needs access to the private data members. Any source file that does not implement the `Speller` class only needs to know the size of the class (just like application using the Simple Spell library). Thus, unless `__speller_impl` is defined all the other source files will see is `void * impl`. Any source file that defines part of the spell checker includes an alternative header file `speller_impl.hpp` which defines the `SugsData` class, the `__speller_impl` preprocessor

```
class Speller
#ifdef __speller_impl
  : fix_size(0)
#endif
{
#ifdef __speller_impl
  ...
  SugsData sugs_data;
#else
  void * impl;
#endif
public:
  Speller();
  void init(Language * lang, Dictionary * main); // will take ownership of both
  bool check(const char *);
  void add_to_session(const char *);
  Suggestions * suggest(const char *); // result only valid to next call to suggest
  ~Speller();
private:
  Speller(const Speller &); // no copy
};
```

Figure 6.8. The `Speller` class using the pimpl idiom.

macro and then includes `speller.hpp`, for example:

```
struct SugsData {...};

#define __speller_impl
#include "speller.hpp"
#undef __speller_impl
```

Figure 6.9 shows the new `Session` class definition. Since, unlike like the `Speller` class, some of the private data members are used by the application (via inline functions) we can not use the pimpl idiom. Instead we fix the size of the class to be just large enough to include the pointer-to-vtable and the private data members used by inline functions.

To make the header file more readable we define a few preprocessor macros that are defined differently depending on whether ZL is being used as follows:

```
class Session
  FIX_SIZE(sizeof(void *) /* vptr */ + sizeof(unsigned) * 2
           + sizeof(char *) + sizeof(void *))
  VTABLE_SLOTS(16) {
protected:
  unsigned misspelled_start;
  unsigned misspelled_stop;
  char * word;
#ifdef __zl
  // other private data members
#else
  void * impl;
#endif
  void reset_line();
public:
  Session(Speller * sp);
  virtual void new_line(const char *);
  virtual bool next_misspelling();
  virtual void replace(const char * new_word);
  const char * misspelled_word() {return word;}
  unsigned misspelled_offset() {return misspelled_start;}
  unsigned misspelled_len() {return misspelled_stop - misspelled_start;}
  virtual ~Session();
#include "vtable_pad-Session.inc"
};
```

Figure 6.9. Improved `Session` class to support future enhancements without breaking binary compatibility.

```
#ifdef __zl
#  define FIX_SIZE(size) :fix_size(size)
#  define VTABLE_SLOTS(size) :vtable_slots(size)
#else
#  define FIX_SIZE(size)
#  define VTABLE_SLOTS(size)
#endif
```

In addition, and unlike the `Speller` class, the `Session` class has a vtable; thus, we also fix the size of the vtable so we can add new methods without breaking binary compatability. However, since we also need to match the GCC ABI and provide a header file that can be used with GCC we need to use a method slightly more complicated than the method described in Section 5.2 in which we used the same `fix_size` macro on the vtable class. The problem is that we need to fix the size of the vtable in GCC's eyes but we can not just provide a dummy character array as we can not directly specify what goes into the vtable, and even if we could the application needs to access the vtable for virtual dispatch. But we can still fix the size by including dummy virtual methods, which is what is included in the file `vtable_pad-Session.inc`:

```
#ifndef __zl
virtual void * dummy__0001_();
virtual void * dummy__0002_();
...
virtual void * dummy__0011_();
#endif
```

Thus, to create the `vtable_pad-Session.inc` file we use a specialized macro designed specially for the vtable class. This macro will, as a side effect, write out a file with the correct number of dummy methods to be used by GCC. In addition, since we are using a specialized macro, we can also allow the user to specify the size in terms of available slots for virtual methods rather than raw size. Thus we use `:vtable_slots` instead of `:fix_vtable_size`.

Since using dummy methods is the only viable method to fixing the size of the vtable with GCC we need to plan ahead and make sure we reserve enough slots to allow for future expansion. We choose to use 16, but we could easily make that larger since the vtable is only allocated once per class (as opposed to once per class instance) and thus does not waste a lot of memory.

The `SessionWFilters` class, shown in Figure 6.10, gets similar treatments except that we fix the size to 0 (and thus effectively use the pimpl idiom) since none of the private data members need to accessed by the application.

Since we already decided that we will not match the GCC ABI for the extension interface we will use the more complicated ABI described in Section 5.3 for the filter class so we don't have to worry about reserving enough vtable slots ahead of time. For reference the the new definition is included in Figure 6.11.

While we changed the ABI from the orignal we have not made any changes to the API, thus we can reuse the same application after a simple recompile.

```
class SessionWFilters : public Session
  FIX_SIZE(0) VTABLE_SLOTS(16)
{
#ifdef __zl
  // private data members
#else
  void * impl;
#endif
public:
  SessionWFilters(Speller * sp);
  virtual SessionWFilters & add(Filter *);
  virtual void new_line(const char *);
  ~SessionWFilters();
#include "vtable_pad-SessionWFilters.inc"
};
```

Figure 6.10. Improved `SessionWFilters` class.

```
extern "C++" : "better"
class Filter {
public:
  Filter();
  virtual void filter(char * line) = 0;
  virtual ~Filter();
  Filter * next;
};
```

Figure 6.11. The `Filter` class using an enhanced ABI.

6.7 A Simple Spell Checker, Version 2

Now that we have defined a better ABI we can make two key enhancements with minimal effort: 1) add a personal dictionary and 2) better support filters with state.

The changes made for the first enhancement are identical to the ones we made in Section 6.5, except that we no longer need any of the tricks in that section as we already fixed the size. All we need to is add the necessary private data members and methods.

So far the filters we have added are stateless, that is they they do not need to maintain any state between lines. But most useful filters will need to maintain sort of state between line; for example, a filter to only check the comments a C or C++ source file will need to know if the previous line started a C style comment. The current API will support such filters as long as only a single document is checked and the lines are checked in sequential order. However, it is sometimes necessary to recheck the same document and often useful to check more than one document without having to create a new session.

Thus, to better support stateful filters we add a new virtual method to the `Session` and `Filter` class, `reset()`, which simply resets the state. Since not all filters need to implement this method we provide a no-op default implementation.

Adding the `reset` method to `Session` would normally break binary compatability since it will change the offset of any derived classes. In particular, in `SessionWFilters`, the offset of the `add` method will change. Thus, if an application calls the `add` using the new `SessionWFilters` class with a header file for the old implementation, the wrong method will be called. As we already discussed, without planning ahead and reserving spots there is little we could have done to avoid this problem while still using the GCC ABI. Fortunately, we did plan ahead and adding the new method does not cause a problem.

To take advantage of the new filter API we added a new filter to our sample application that simple checks all comments in a C or C++ source file and ignores the rest.

To verify that we indeed maintained binary compatability, we linked the original application (before the changes in this section) with the new library without recompiling and verified that everything worked. In addition we tried adding the `reset` method without fixing the size of the vtable and verified that the wrong method was called as predicted.

6.8 An Opportunity for an Even Better ABI

The enhanced ABI we used for the `Filter` class (from section 5.3) goes a long way towards preserving binary compatibility. The new ABI will avoid changing the offsets of any virtual methods of derived classes when adding new methods to the base class. But unfortunately this is still not enough to allow us to use filters compiled with the old ABI with the new ABI, at least without some extra care. That is, we can use a filter from the old ABI as long as the new `reset` method is never called. If we do try to call the reset method on a filter with the old ABI, the application will crash. The problem is that filters compiled with the old ABI will still use the original vtable since they are created statically when the application starts. Hence, the application will crash since the original vtable does not contain that slot.

A better ABI, which could be implemented in ZL, could avoid this problem by dynamically creating the vtable so that all derived classes will use the vtable for the `Filter` class of the new ABI regardless of which ABI they where originally compiled with. This change will allow new virtual methods to be called without a problem provided that they have a default implementation. If they do not (i.e., they are pure virtual) then source code compatability will also be broken and hence there is no point in trying to maintain binary compatibility.

6.9 Comparison to a Real Spell Checker: Aspell

Simple Spell is modeled after a real spell checker, Aspell [2]. In many ways the interface to Simple Spell mirrors that of Aspell. Of course, Aspell is far more complex than Simple Spell, with Aspell containing around 30,000 lines of code and Simple Spell containing between 1,100 and 1,700 lines of code (depending on which version) (see Table 6.1).

To mitigate ABI compatibility problems Aspell does not expose a C++ interface. Instead, a Perl script is used to generate the C interface. The Perl script is around 1,900 lines of code, with around a 1,000 line input file. The Perl script generates around 3,000 lines of code, of which 400 lines consist of code that is now manually maintained. An additional 1,900 lines of manual interface code is also used in the C interface (not including the 400

Table 6.1. Approximate lines of code of the various versions of Simple Spell and Aspell.

Spell Checker Version	Section	Lines Of Code
Simple Spell (Initial Version)	6.1	1,100
Simple Spell w/ Email Filter	6.4	1,300
Simple Spell w/ Personal Dictionary	6.5	1,300
Simple Spell w/ Better ABI	6.6	1,600
Simple Spell, Version 2	6.7	1,700
Aspell	-	30,000

lines once generated with the Perl script). The large line count for the interface code reflects the fact that Aspell is a moderately complex program, and also that the bridge between the internal C++ interface and external C interface is rather involved; for example, it includes support for managing memory and conversion of the input and output from one encoding (such as UTF-8) to Aspell's internal 8-bit encoding.

Due to its use of advanced C++ features (such as templates) Aspell is currently unable to compile under ZL. Had it been able to compile, the bridge code (from the internal C++ ABI to the external C ABI) could be written using ZL; the information in the interface file could become part of the class, and a modified class macro can be used to extract it. It remains to be seen if the end result will be any simpler. More importantly, by using many of the techniques outlined in this chapter, it will likely be possible to directly expose a stable C++ that will not change between different releases of Aspell.

CHAPTER 7

USING ZL

As the main ZL compiler compiles to a C-like language and does not create executables directly, a driver script is provided to create the executable. The driver is designed to act as a drop in replacement for GCC. For example,

```
zlc main.zl file1.cpp file2.zlp -o main
```

compiles the pure ZL source file `main.zl`, the C++ source file `file1.cpp`, and the ZL source file `file2.zlp` into an executable `main`. Each file is compiled slightly differently based on the extension as ZL has different modes for C, C++, and ZL source files. In addition C, C++, and ZL source files with the `.zlp` extension are run through the C preprocessor to handle includes and other token-level macros that the higher-level ZL macro processor can not handle. Pure ZL files with `.zl` extension are not preprocessed.

The rest of this chapter gives the additional details of ZL of interest to the macro or tool implementer. The rest of the implementation details worth mentioning are given in the next chapter.

7.1 Classes and User Types

A user type (see 4.1), which is ZL's minimal notion of a class, consists of two parts: a type, generally a `struct`, to hold the data for the class instance, and a *module*, which is collection of symbols for manipulating the data.

As an example,

```
class C { int i;
          int f(int j) {return i + j;} };
```

expands to:

```
user_type C {
  struct Data {int i;};
  associate_type struct Data;
  macro i (:this ths = this) {(*(C *)ths)..i;}
  macro f(j, :this ths = this) {f`internal(ths, j);}
  int f`internal(C * fluid this, int j) {return i + j;}
}
```

and creates the class `C`.

To allow user types to behave like classes, member-access syntax gets special treatment. For example, if `x` is an instance of the user type above, `x.i` calls the `i` macro in the `C` module, and it passes a pointer to `x` as the `this` keyword argument. This protocol allows `x.i` to expand to something that accesses the `x` field of the underlying struct, which can be done using the special syntax `x..i`. Thus, `i` effectively becomes a data member of `x`. Methods can similarly be defined. For example, `x.f(12)` calls the `f` macro with one positional parameter and the `this` keyword argument.

The default value for the `this` keyword argument is necessary to support the implicit `this` variable when data members and methods are accessed inside method definitions. The function `f`internal`, which implements the `f` method, demonstrates this. (The `` `internal `` simply specifies an alternative namespace for the `f` symbol so that it does not conflict with the `f` macro.) The first parameter of the function is `this`, which puts the symbol into the local environment. When `i` is called inside the function body the `this` keyword argument is not supplied, since we are not using the member access form. Therefore, the `this` keyword argument defaults to the `this` specified as the default value, which binds to the `this` in the local environment. The `fluid` keyword (see 7.3.7) is necessary to make the `this` variable visible to the `i` macro; with normal hygiene rules, binding forms at the call site of a macro are invisible, as symbols normally bind to whatever is visible where the macro was defined.

User types can also be declared to have a subtype relationship. The declaration specifies a macro for performing both casts to and from the subtype. Subtypes are used to implement inheritance. For example the class:

```
class D : public C { int j; };
```

expands to something like:

```
user_type D {
  import C;
  struct Data {struct C::Data parent; int j;};
  associate_type struct Data;
  macro _up_cast (ths) {&(*ths)..parent;}
  macro _down_cast (other) {(D*)other;}
  make_subtype C _up_cast _down_cast;
  macro j (:this ths = this) {(*(D*)ths)..j;}
}
```

New symbols defined in a module are allowed to shadow imported symbols, so the fact that there is also a `Data` in `C` does not create a problem. Also, note that there is no need to redefine the data member and method macros imported from `C`, since the existing ones will work just fine. They work because the class macro makes sure that the `ths` macro parameter is cast to the right type before anything is done with it. For example, if `y` is an instance of the type `D`, then `y.i` expands to `(*(C*)&y)..i`. When ZL tries to cast `&y` to `C*`, the `D::_up_cast` macro is called and the expression expands to `((*&(*&y)..parent))..i`, which simplifies to `y..parent..i`. Method calls expand similarly, except that the cast is implicit when the `ths` macro parameter is passed into the function.

If a class contains any virtual methods, then a vtable is also created. The macro that implements the method then looks up the function in the vtable instead of calling it directly. For example, if `f` was a virtual function in the class `C`, then the macro for `f` would look something like:

```
macro f(j, :this ths = this) {_vptr->f(ths, j);}
```

where `_vptr` is a hidden member of the class that contains a pointer to the virtual table. The vtable is also a class, so to implement inheritance with virtual methods a child's vtable simply inherits the vtable of the parent. To override a method, the constructor for the child's vtable simply assigns a new value to the entry for the method's function pointer.

7.2 Pattern-Based Macros and Lexical Extensions

Figure 7.1 shows a pattern-based macro (see 4.2) that iterates over an STL-like container. To print the contents of `con`, a container of integers, one would use:

```
foreach(x, con, {printf("%d\n", x);});
```

```
macro foreach (VAR, WHAT, BODY) {
  typeof(WHAT) & what = WHAT;
  typeof(what.begin()) i=what.begin(), e=what.end();
  for (; i != e; ++i) {
    typeof(*i) & VAR = *i;
    BODY;
  }
}
```

Figure 7.1. Macro that iterates over an STL-like container.

7.2.1 Extending the Parser

The syntax of the `foreach` macro in Figure 7.1 is a bit ugly. It would be nice if we could instead write something like:

```
foreach (x in con) printf("%d\n", x);
```

which does not have the shape of a function call. ZL lets us do this by modifying[1] the `STMT` production in the grammar for the parser (from raw text to syntax objects) to recognize the new form:

```
<foreach> "foreach" "(" {ID} "in" {EXP} ")" {STMT}
```

In this grammar, anything between `{}` becomes a subpart of the syntax object that is named between the `<>`. We must pair this modification with a macro for the new syntax form. The definition of the new `foreach` macro is identical to the function-call one except that `smacro` is used instead to declare that the macro works with a syntax object produced by the parser.

Support for both styles of macros (function call and syntax) is important, because not every macro warrants support in the parser. For example, since the `or` macro from Section 4.2 has limited usefulness, it probably does not warrant adding a new operator. Furthermore, function-call macros are typically sufficient in generating boilerplate code. In contrast, new general-purpose forms typically merit a parser extension.

---

[1] More modular lexical extensions that do not requiring modifying the full grammar is future work. (See Section 11.6.)

7.2.2 The Parser

The ZL grammar is specified through a PEG [35], but with a few extensions to the usual PEG notation, and a Packrat [34] parser is used to convert strings of characters to syntax objects. A simplified version of ZL's initial grammar is shown in Figure 7.2. For readers not familiar with PEGs, the two most important things to note are that PEGs work with characters rather than tokens, and the `/` operator defines a prioritized choice. A prioritized choice is similar to the `|` operator used in Backus-Naur Form, except that it *unconditionally* uses the first successful match. For example, given the rule "`A = 'a' / 'ab'`" the string `ab` will never match because the first choice is always taken. The PEG specification more closely resembles regular expression syntax (as used in `grep`) than it does Backus-Naur Form. The `()`, `[]`, `?`, `*`, `+`, and `_` (otherwise known as `.`) operators are all used in the same manner as they are in regular expressions. Anything between single quotes is a literal string. The double quote is like the single quote, except that special rules make them behave similarly to tokens. For example, `"for"` will match the `for` in `for(`, but it will not match the prefix of `foreach`. The `{}` and `<>` are extensions to the standard PEG syntax and are used for constructing syntax objects in the obvious ways. The special `<<mid>>` operator and `MID` production are explained later in Section 8.2.

7.2.3 Built-in Macros

The grammar serves to separate individual statements and declarations, and to recognize forms that are convenient to recognize using a Packrat parser. As such, it creates syntax objects that need additional processing before they can be compiled into an AST. The expander has several built-in macros for this purpose: `stmt`, `exp`, `()`, `[]`, and `{}`.

The `stmt` macro recognizes declarations and expressions. It first tries the declarations expander, which is a handwritten parser designed to deal with C's idiosyncratic syntax for declarations. If the declarations expander fails, then the expression expander is tried, which is an operator-precedence parser [32]. The `exp` macro is like the `stmt` macro, but only the expression expander is tried.

The macros `()`, `[]`, and `{}` are used for reparsing strings. The `()` and `[]` macros reparse the string as an expression using the `EXP` production in the grammar, where as the `{}` generally reparses the string as a block using the `BLOCK` production.

```
TOP = <top> SPACING {STMT}+;

STMT = <<mid PARM>> {MID} ";"
     / <if> "if" "(" {EXP} ")" {STMT} ("else" {STMT})?
     / <while> "while" "(" {EXP} ")" {STMT}
     / <break> "break" ";"
     / <return> "return" {EXP} ";"
     / {BLOCK}
     # other statements ...
     / <stmt> ({TOKEN_}+ {PAREN} {BRACE} / {TOKEN}+ ";");

EXP = <exp> {TOKEN}+;

BLOCK = <block> "{" {STMT}* "}";

TOKEN_ =  <<mid PARM>> {MID} / {BRACK} / {CONST} /
          {ID} / {SYM};
TOKEN = TOKEN_ / PAREN;

PAREN = <()> "(" {RAW_TOKEN*} ")";
BRACE = <{}> "{" {RAW_TOKEN*} "}";
BRACK = <[]> "[" {RAW_TOKEN*} "]";

CONST = <f> ...  / <l> ... / # float, numeric literal
        <s> ... / <c> ...    # string, character

ID = <<mid>> {MID} / {[@$\a_][\a_\d]*} SPACING;

SYM = {'...' / '==' / '+' / ...} SPACING;

RAW_TOKEN = STRING / CHAR / SYM / BRACE / PAREN /
            BRACK / COMMENT / [^\)\]\}];

STRING = '"' ('\\'_/[^"])+ '"' SPACING;
CHAR   = '\'' ('\\'_/[^'])+ '\'' SPACING;

SPACING = [\s]* COMMENT?;

COMMENT = ...;
```

Figure 7.2. Simplified PEG grammar.

7.3 Macro API

Figure 7.3 shows a procedural-macro (see 4.4) version of the `foreach` macro, which returns an error message if the container does not contain the `begin` or `end` method. This section gives the details of procedural macros and its API. The API has both a class-like form (see 5.4) and a procedure form; this section presents the class-like form. The mapping from the class API to the raw API is straightforward. The general scheme is that the object name prepends the method in all lower case with an underscore separating it from the method name. The object is then passed in as the first parameter. For example, the method:

```
Syntax * Match::var(UnmarkedSyntax * var);
```

becomes

```
Syntax * match_var(Match *, UnmarkedSyntax * var)
```

7.3.1 The Syntax Object

The API for syntax objects (see 4.3) is lised in Figure 7.4. There are two syntax-objects types, `UnmarkedSyntax` and `Syntax`. The difference between the two is the first represents a syntax object that has not been marked (see 4.4) yet, while the second one has. A `Syntax` object will automatically convert to a `UnmarkedSyntax`. But in order to go from

```
Syntax * foreach (Syntax * syn, Environ * env) {
  Mark * mark = new_mark();
  Match * m = match_args(0, syntax(VAR,CON,BODY), syn);
  Syntax * what = m->var(m, syntax CON);
  if (!symbol_exists(syntax begin,what,mark,env) && ...)
    return error(what,
                 "Container lacks begin or end method.");
  UnmarkedSyntax * repl = syntax {
    typeof(CON) & what = CON;
    typeof(what.begin()) i=what.begin(), e=what.end();
    for (; i != e; ++i) {typeof(*i) & VAR = *i; BODY}};
  return replace(repl, m, mark);
}
make_syntax_macro foreach;
```

Figure 7.3. Version of `foreach` that returns a helpful error message if the container does not contain the `begin` or `end` methods.

Type `UnmarkedSyntax`

Type `Syntax`, *subtype of* `UnmarkedSyntax`, with **methods**:

```
Syntax * num_parts(unsigned)
Syntax * part(unsigned)
Syntax * flag(UnmarkedSyntax *)
bool simple()
bool eq(UnmarkedSyntax *)
Syntax * stash_ptr(void *) (static method)
void * extract_ptr()
```

Figure 7.4. Syntax object API.

`UnmarkedSyntax` to `Syntax` the syntax object needs be marked, which is generally done via `replace`.

Internally `UnmarkedSyntax` and `Syntax` are the same type. The distinction in the API is to avoid invalid use of unmarked syntax objects.

A syntax object consists of one or more parts, and optional flags. The first part has special meaning and is used to identify the syntax, provided that it is simple. A *simple* syntax object is basically[2] a syntax object with just one part, and no flags. Internally it is represented slightly differently. Parts other than the first are considered arguments.

Syntax objects can also have any number of optional flags. A flag is a named argument and is retrieved by name, rather than position. A flag itself is just a normal syntax object with the first part used to name the flag. Flags can be tested for existence using the `Syntax`'s `flag` method (which returns `NULL` if the flag does not exist) or matched with the match family of functions (see 7.3.3). Flags are primarily used when parsing declarations and can be created in macros by using the `raw_syntax` primitive. For example the following syntax object:

```
(... :flag1 :(flag2 value2))
```

contains two flags, where `flag1` is a flag without any value associated with it while `flag2` is a flag with a value. Flags can also be passed into function call macros in which are just another name for the already described keyword arguments.

---

[2] This is an over simplification since "foo" and "(foo)" are not the same. The first is considered simple while the second is not.

Syntax objects can also contain other types of objects embedded within them. A syntax object of such form is considered an *entity*. The most common types of objects are parsed syntax either in the form of an AST node or a symbol. However, it is also possible to embed arbitrary objects such as pointers in a syntax object using the `stash_ptr` and `extract_ptr` methods. These methods are most commonly used in combinations with Symbol properties, which will be described in 7.4.4.

Sometimes it is useful to get information on the syntax object without having to use `match`. For this ZL provides a number of methods to directly access the syntax object and get basic information. The `part` and `num_parts` method can be used for direct access. The `eq` and `simple` method can be used to get basic properties on the syntax object. The `eq` method tests if the syntax object is equal to another, taking into account that the first one may be marked. The `simple` method tests if the syntax object is simple as previously described.

7.3.2 The Syntax List

A *syntax list* is a syntax object whose first part is a `@`. It represents a list of syntax objects (which can include flags). Lists have the effect of being spliced into the parent syntax object.

Syntax lists can be used as values for macro identifiers, in which case the results are spliced in. Macros can return syntax lists, but the results are not automatically spliced in. Rather when a list of elements is parsed any `@` are flattened as the list is read in. It is an error to return a syntax list in a nonlist context.

The `SyntaxList` API is shown in Figure 7.5. Syntax lists are created using the `new_syntax_list` function. Elements are then appended to the list using the `append` or `append_flag` method. The `empty` method returns true if the list has 0 elements. The `elements` method is used to iterate through the elements and return a `SyntaxEnum`. The `next` method of `SyntaxEnum` returns the next element in the list or NULL if there is none, while the `clone` method returns a copy of the `SyntaxEnum`.

Type `SyntaxList`, *subtype of* `Syntax`, with **constructor**:

```
SyntaxList * new_syntax_list()
```

and **methods:**

```
int empty()
void append(Syntax *)
void append_flag(Syntax *)
SyntaxEnum * elements()
```

Type `SyntaxEnum` with **methods**:

```
Syntax * next()
SyntaxEnum * clone()
```

Figure 7.5. Syntax list API.

7.3.3 Matching and Replacing

Figure 7.6 lists the API for matching and replacing (see 4.4). The `match`, `match_args`, and `replace` functions have already been described. The `var` method is identical to the previously described `match_var` function. The `varl` method is like `var` except that it returns an an enumeration for iterating through the elements of a syntax object that is also a list. The fact that it results an enumeration rather than a list is deliberate, since syntax lists are mutable objects, and the results from a match are not.

When it is necessary to build syntax directly from syntax objects, the `match_local` function provides a convenient way to do so. It takes in a match object and a list of syntax objects, terminated by `NULL`. It will assign a numeric match variable in the form of `$NUM`

Type `Match` with **methods**:

```
Syntax * var(UnmarkedSyntax *)
SyntaxEnum * varl(UnmarkedSyntax *)
```

and **related functions**:

```
Match * match(Match * prev, UnmarkedSyntax * pattern, Syntax * with)
Match * match_args(Match *, UnmarkedSyntax * pattern, Syntax * with)
Match * match_local(Match *, ...)
```

Callback function:

```
Syntax * replace(UnmarkedSyntax *, Match *, Mark *)
```

Figure 7.6. Match and replace API.

with the first one being `$1`.

7.3.4 Match Patterns

A pattern to be matched against is expected to either be a simple list of the form `syntax (a, b, ...)` or fully parsed, i.e., created with `raw_syntax`. The difference is that pattern variables matched with the former will need to be reparsed while patterns variables matched with latter do not.

The `syntax ()` form is designed to be used when matching parameters passed in via a function-call macro. The pattern contains a list of the following (with some restrictions on order):

- *ID*
- *ID* = *VALUE* – must be after all plain ID's
- `@` – can only appear once
- `@`*ID* – must be last
- :*FLAG*
- :*FLAG ID*
- :*FLAG ID* = VALUE

ID matches a normal parameter. The second item, "*ID* = *VALUE*", is used for giving parameter default values if they are omitted. A _ can be used any place an identifier will be used when the value is irrelevant. Parameters can also be optional if they are after the special `@` instruction, in which case they will simply be omitted from the match list. The `@`*ID* form will match any remaining parameters and store them in a syntax list. Flags can also be matched with any of the :*FLAG* forms. Flags, in the current implementation, are always optional; however, any matched flags will not appear in the syntax list matched with `@`*ID*.

A pattern can also be specified in `raw_syntax` form, which is designed to be used with syntax macros. In the `raw_syntax` form a pattern can represent anything that a match list can. In addition, it is possible to match the subparts of an expression using `(pattern (WHAT ...))`. For example, to match the list of declarations inside of a class body which

is represented as `(class foo ({...} decl1 decl2))` into the pattern variable `body`, the `(_ _ (pattern ({...} @body)))` pattern can be used.

It is also possible to use the `raw_syntax` form with function-call macros; however, when doing so it is important to know that the macro parameters are not parsed. For example if `f` is a function-call macro, the parameter of the call `f(x+2)` is passed in as `(parm "x+2")`. When using the `syntax` forms for matching, ZL's normal parsing process (see 4.3, 8.2) parses the string at the right time. But the `raw_syntax` form skips this step. Thus, it it necessary to manually instruct ZL to parse the parameter passed in by using `(reparse ID)`. For example, to match the parameter in the `f` macro above use:

```
match_args(..., raw_syntax((reparse ID), ...))
```

7.3.5 Creating Marks

Marks (see 4.4) are used to implement lexical scope, and the API is listed in Figure 7.7. The `new_mark` primitive is actually a macro that calls the callback function `new_mark_f` and uses the primitive `environ_snapshot()` to capture the environment.

7.3.6 Controlling Visibility

The `get_context` and `replace_context` functions, shown in Figure 7.8, are used to bend hygiene in a very similar fashion to `datum->syntax-object` in the `syntax-case` expander [23]. For example, a macro defining a class needs to create a vtable that is accessible outside of the macro creating the class. The `get_context` function gets the context from some symbol, generally some part of the syntax object passed in, while

Type `EnvironSnapshot` with **related syntax form**:
`environ_snapshot()` — returns `EnvironSnapshot *`

Type `Mark` with **related function**:

```
Mark * new_mark_f(EnvironSnapshot *)
```

and **macros**:

```
macro new_mark(es = NULL) {new_mark_f(es ? es : environ_snapshot();}
macro new_empty_mark() {new_mark_f(0);}
```

Figure 7.7. Mark API.

Type `Context` with **related functions**:

```
Context * get_context(Syntax *)
Syntax * replace_context(UnmarkedSyntax *, Context *)
```

Figure 7.8. Visability API.

`replace_context` replaces the context of the symbol with the one provided. For example, code to create a symbol `_vtable` that can be used later might look something like:

```
...
Match * m = match_args(0, raw_syntax (name ...), p);
Syntax * name = m->var(m, syntax name);
Context * context = get_context(name);
Syntax * _vtable = replace_context(syntax _vtable, context);
...
```

Here `name` is the name of the class that is passed in as `m`. The `name` symbol is extracted into a syntax object so that it can be used for `get_context`. The `replace_context` function is then used to put the symbol `_vtable` in the same context as `name`. Now `_vtable` will have the same visibility as the `name` symbol, and thus be visible outside the macro.

7.3.7 Fluid Binding

The `get_context` and `replace_context` functions are one way to bend hygiene. The other is to use `fluid_binding`, which allows a variable to take its meaning from the use site of a macro rather than the macros's definition site, in a similar fashion to `define-syntax-parameter` in Racket [31, 16].

A prime example of the need for `fluid_binding` is the special variable `this` in classes. Variables in ZL are lexically scoped. For example, the code:

```
int g(X *);
int f() {return g(this);}
int main() {X * this = ...; return f();}
```

will not compile because the `this` defined in `main` is not visible in `f`, even though `f` is called inside `main`. However, if the `this` variable was instead dynamically scoped, the `this` in `main` would be visible to `f`.

Normal hygiene rules preserve lexical scope in a similar fashion, such that:

```
int g(X *);
macro m() {g(this);}
int main() {X * this = ...; return m();}
```

will also not compile. Attempts to make this work with `get_` and `replace_context` will not compose well [16]. What is really needed is something akin to dynamic scoping in the hygiene system. That is, for `this` to be scoped based on where it is used when expanded, rather than where it is written in the macro definition. This can be done by marking the `this` symbol as `fluid` using `fluid_binding` at the top level and then using `fluid` when defining the symbol in local scope. For example:

```
fluid_binding this;
int g(X *);
macro m() {g(this);}
int main() {X * fluid this = ...; return m();}
```

will work as expected. That is, the `this` in `m` will bind to the `this` in `main`.

7.3.8 Partly Expanding Syntax

In complex syntax macros, it is often necessary to decompose the parts passed in. However, in most cases, those parts are not yet expanded; thus it is necessary to expand them first.

For instance if it was necessary to to decompose the syntax for the container passed into `foreach` in Figure 7.3, the syntax object would need to be expanded first, as at the point the macro was called, the container is likely still represented as a generic `exp`, which is just a list of tokens. For example, if the container were the identifier `c`, the syntax object for the container would be `(exp c)` instead of `(id c)`. To support this decomposition ZL provides a way to partly expanded a syntax object in the same way it will internally; the API is shown in Figure 7.9.

The `pos` parameter tells ZL what position the syntax object is in; the values of the `Position` enum can be bitwise or'ed together. This parameter will affect how the expansion and, if necessary, reparsing is done. Common values are `TopLevel` for declarations, `StmtPos` for statements, and `ExpPos` for expressions. The `Environ` parameter is the environment as passed into the macro.

Callback functions:

```
Syntax * partly_expand(Syntax *, Position pos, Environ *)
SyntaxEnum * partly_expand_list(SyntaxEnum *, Position pos, Environ *)
```

and **enum** `Position` with **possible values**:

`NoPos`, `OtherPos`, `TopLevel`, `FieldPos`, `StmtDeclPos`, `StmtPos`, `ExpPos`

Figure 7.9. Expander API.

If the parts of a syntax object represent a list of some kind, it is best to use `partly_expand_list`. The function `partly_expand_list` is like `partly_expand`, except that it expects a list of elements in the form of an `SyntaxEnum`, and it automatically flattens any Syntax Lists (ie `@`) found inside the list. The elements of the list are expanded as they are iterated through, rather than all at once when the function is called.

7.3.9 Compile-Time Reflection

Often it is necessary to do more than just decompose syntax. Sometimes, it is necessary to get compile-time information on the syntax objects or the environment itself—for example, to get numerical value of an expression as was done in with `fix_size` in Section 5.1 or to check if a symbol exists as is done in `foreach` in Figure 7.3. Figure 7.10 shows some of the available API functions for compile-time reflection.

The `ct_value` function (which was used in the `fix_size` example) takes a syntax object, expands the expression, parses the expansion, and evaluates the parsed expression as an integer to determine its value. An error is thrown if the expression passed in is not a compile time constant.

To see if a symbol exists in the current environment or an object that is a user type (as

Callback functions:

```
unsigned ct_value(Syntax *)
bool symbol_exists(UnmarkedSyntax * sym, Syntax * where,
                   Mark *, const Environ *)
Environ * temp_environ(const Environ *)
Syntax * pre_parse(Syntax *, Environ *)
```

Figure 7.10. Compile time reflection API.

was done in the `foreach` example), the `symbol_exists` function can be used. The `first` argument is the symbol to check for. The `second` argument is the user type to check that the symbol exists in; if it is `NULL` then the current environment will be checked instead. The third argument provides the context in which to look up the current symbol, and finally the last argument is the environment to use.

Sometimes in order to get compile-time information it is necessary to add additional symbols to the environment. For this the `temp_environ` and `pre_parse` functions are used, as was done in the `fix_size` macro. The `temp_environ` function creates a new temporary environment while `pre_parse` parses a declaration just enough to get basic information on it, and then adds it the the environment. The creation of a temporary environment avoids affecting the outside environment with any temporary objects added with `pre_parse`.

7.3.10 Misc API Functions

Sometimes it is necessary to create syntax on the fly, such as creating syntax from a number that is computed at run time. The `string_to_syntax` function, shown in Figure 7.11, converts a raw string to a syntax object. The string passed in is the same as given for the `syntax` form, which can be specified at run time.

The `syntax_to_string` function does the reverse, which is primarily useful for checking an identifier for a literal value. It is also useful for debugging to see the results of a complex macro. However, for large syntax objects the `dump_syntax` function is more efficient. For complex syntax objects the output of both functions is designed to be human readable and as such the output is not suitable for reparsing with `string_to_syntax`.

The `error` function is used to return an error condition as in done with `foreach` in

Callback functions:

```
UnmarkedSyntax * string_to_syntax(const char *)
const char * syntax_to_string(UnmarkedSyntax *)
void dump_syntax(UnmarkedSyntax *)
Syntax * error(Syntax *, const char *, ...)
```

Figure 7.11. Misc API functions.

Figure 7.3. It creates a syntax object that results in an error when it is parsed. The first argument is used to determine the location where the error will be reported; the location associated with this syntax object is used as the location of the error.

7.4 Procedural Macro Implementation and State Management

In order to use procedural macros effectively, it is necessary to know a little bit about how they are implemented. This section gives the details on how procedural macros are implemented, the use of macro libraries, and how to share state between procedural macros.

7.4.1 The Details

The current ZL compiler does not contain an interpreter; thus procedural macros are compiled and then dynamically linked into the compiler when the macro is first used. A simple dependency analysis is done so that any components that the procedural macro depends on (and are not already compiled and linked in) are also compiled at the same time.

In addition, ZL determines the role of each function as for run-time or compile-time only to avoid included macro related functions in the executable. A compile-time only function is any function that uses part of a macro API, or one that depends on a function which does.

The dependency analysis that determines which code to include when a procedural macro is first used is separate from the dependency analysis used to determine a role. Thus, it is possible for a function to be used at both run-time and compile-time if the function is used by both a normal (i.e., run-time) function and a compile-time only function. Such a function will be considered a run-time function even though it is also used at compile time.

7.4.2 Macro Libraries

Since the compilation of a complicated procedural macro can take a decent amout of time, ZL also provides a mechanism for precompiling macros ahead of time via macro libraries. A macro library is similar to a normal library, except that the code is loaded while compiling the program, instead of during the programs execution.

A macro library is a collection of code compiled with the `-C` option. The compilation creates a shared library with the `-fct.so` extension; for example, if the code for the library was contained in the file `lib.zl`, the shared library will be called `lib-fct.so`. The macro library is then used by importing the same file (used to create the library) using the `import_file` primitive. Importing will: 1) parse enough of the macro library code to get the function prototypes and related information; and 2) load the related shared library. A header file can also be provided (with an extension of `.zlh`), which will be read in instead of the full macro code.

Normally, when `new_mark()` (which uses the `environ_snapshot()` primitive) is used, the environmental snapshot is taken at the place in the code where the syntax is used. (Basically, `environ_snapshot()` gets replaced with a pointer the the current environment as the procedural macro is being parsed.) Unfortunately, ZL does not have the ability to serialize the environment, which means a snapshots can only be taken for code that is compiled in the same translation unit (also known as the compilation unit). This creates a problem when a procedural macro is compiled into a library. To work around this problem the user can declare that the environmental snapshot is taken where the macro is declared, rather then where `environ_snapshot()` is used, by adding `:w_snapshot` to `make_macro`, for example:

```
make_syntax_macro foreach :w_snapshot
```

Since, unlike the function body, the `make_macro` declaration is always read as the program is being compiled, this ensures that there is always a point where the snapshot can be taken. In the rare cases when this strategy will not work, it is possible to store a snapshot of an environment in a variable. For example, if:

```
EnvironSnapshot * prelude_envss = environ_snapshot();
```

is found in a header file, than ZL will ensure that the value of global variable `prelude_envss` is a pointer to an environmental snapshot in the current compilation unit. Within the macro library, this variable can then be used with an alternative form of `new_mark`, which accepts a pointer to `EnvironSnapshot` as its first parameter.

When macro libraries are used no automatic dependency analysis is done; everything included in the macro library is assumed to used at compile-time only. If it is necessary

to use the same code at both compile-time and run-time, special previsions need to be made, such as moving the shared code into a separate file so that it can be linked in at both compile and run time. Linking compile-time only functions into the executable will fail with undefined symbols.

7.4.3 State Management

Macros may maintain global state in one of two ways. The first way is to simply use global variables; any state stored within a global variable will be accessible to any macros used in the same compilation, even if they are compiled and linked in separately. The other way to maintain global state is to store the information inside of a top-level symbol via the use of symbol properties, the details of which are provided in the next section.

Using either method, state is only maintained during within the compilation unit. Separate provisions need to be made to store state between compilations.

7.4.4 Symbol Properties

Any top-level symbol can have any number of *properties* associated with it. The value of the propriety is simply a syntax object. Symbol properties are used extensively by the class macro to store information about the class which is then used by the parent class and when expanding method definitions defined outside of the class.

Figure 7.12 shows the syntax for the `add_prop` primitive used for adding symbol properties. Note that `add_prop` is not an API function; it is part of the syntax returned by the macro. In addition, the `add_prop` primitive is always used in the lower level

Syntax to add properties to existing symbols:

```
(add_prop SYMBOL PROPERTY-NAME VALUE)
```

Syntax to add properties within modules:

```
(add_prop PROPERTY-NAME VALUE)
```

Macro API function to retrieve properties:

```
Syntax * get_symbol_prop(UnmarkedSyntax * symbol, UnmarkedSyntax * prop,
                         const Environ *)
```

Figure 7.12. Symbol properties syntax and API.

s-expression form (i.e., created using `raw_syntax` instead of `syntax`) in order to be able to precisely control the syntax object being added. Such control would not be possible in the higher level syntax due to reparsing.

The three argument form of `add_prop` is used to add properties to already existing top-level symbols. For example the class macro adds the propriety `is_method` to the macro representing methods by using:

```
(add_prop (fun method (. @parms)) is_method true)
```

where `method` and `@parms` are pattern variables. The two argument form of `add_prop` is used within a module or user type to add properties to the module.

To retrieve properties from a symbol the macro API function `get_symbol_prop` can be used. The function will return NULL if the property does not exist for that symbol.

When used in combination with `stash_ptr` and `extract_ptr` arbitrary objects can be stashed away for latter retrieval. For example the class macro uses this to store a pointer to the class used to implement the class in the module for the class. This pointer is then extracted when expanding method definitions defined outside of the class, thus greatly simplifying the implementation.

7.5 ABI Related APIs

This section gives additional procedural macro API components that are important to creating classes and controlling the ABI.

7.5.1 User Type and Module API

Within the class macro it is necessary to get some basic properties on data memeber types and the parent class. In particular it is necessary to determine if the type is a user type with any special methods such as a default constructor or destructor. The API for user types and modules is shown in Figure 7.13.

The constructors `user_type_info` and `module_info` get the corresponding symbol from a symbol name. From a `user_type` it is also possible to get the underlying module using the `module` method.

The `have_*` user type methods are used to check if a data-member type has any special methods. The class macro uses this information when building the corresponding special

Constructors:

```
UserType * user_type_info(Syntax *, const Environ *)
Constructor: Module * module_info(Syntax *, const Environ *)
```

Type `UserType` with **methods**:

```
Module * module()
bool have_default_constructor()
bool have_copy_constructor()
bool have_assign()
bool have_destructor()
```

Type `Module` with **methods**:

```
bool have_symbol(const Syntax *)
```

Figure 7.13. User type and module API.

method. For example if any the data-members have the assign method is is necessary to create an assign method for the class.

7.5.2 User Type Builder

Due to the need to get information about the user type as it is being built, the class macro builds the user type directly and then returns a syntax object with the compiled syntax object embedded directly. The builder API is shown in Figure 7.14.

A new builder is created using `new_user_type_builder`. Components are added

Type `UserTypeBuilder` with **constructor**:

```
UserTypeBuilder * new_user_type_builder(Syntax * name, Environ * env)
```

and **methods**:

```
void add(Syntax *)
Syntax * to_syntax()
bool have_default_constructor()
bool have_copy_constructor()
bool have_assign()
bool have_destructor()
```

and **members** (read only):

```
Environ * env
UserType * user_type
```

Figure 7.14. User type builder API.

using the `add` method. Finally, the `to_syntax` method is used to finalize the user type and return a syntax object with the compiled user type embed within.

The `have_*` methods are used for querying the user type as it is being built. They are needed because, due to overloading, it is difficult for the class macro to determine if a constructor or assignment operator is provided that satisfies the requirements of a copy constructor or copy assignment operator, restively. Thus, after all the methods are added to the user type, the class macro uses these methods to check for the existence of the special methods and can act appropriately.

The user type builder also exposes several read only members. The most important one is the local environment inside the module. This environment is needed when partly expanding class components, for example, in the following code:

```
typedef const char * iterator;
iterator begin();
```

the second line, will not expand correctly unless the `iterator` type is in the environment.

7.5.3 The ABI Switch

Since class layout is a key component to the ABI, a new ABI can be created by extending (or overriding) the class macro and then remapping the `class` syntax object to use the new macro. Another way to define a new ABI is to register the ABI so that it can be used with the ABI switch. The ABI switch is an extension of C++ `extern` with an additional part for the ABI. For example:

```
extern "C++" : "gcc"
class C {...};
```

causes the class `C` to use the "gcc" ABI. In addition to class layout, the ABI switch also controls name managing and other key componets of the ABI, which can differ between compilers.

A class macro is registered with the ABI switch by compiling it into into a macro library and defining the symbols `_abi_list` and `_abi_list_size`. The `_abi_list` variable is an array of `AbiInfo` and `_abi_list_size` is the array size. The struct `AbInfo` is defined as:

```
struct AbiInfo {
  const char * abi_name;
  MangleFun mangler;
  MacroLikeFun parse_class;
  const char * module_name;
  Module * module;
};
```

The `abi_name` member is the name of the ABI, and `parse_class` points to the macro function defining the class. The `mangler` member is part of the mangler ABI and will be described the next section.

Class layout and mangling are two important parts of the ABI. Another important part is the implementation of `new` and `delete`. To support any ABI specific implementations a module name can be provided. Any symbols is this module will shadow any global symbols when the ABI is in effect; thus ABI specific `new` and `delete` macros can be defined. In addition the ABI info is tied to a user type so a class is always allocated and deleted with the class ABIs `new` and `delete`.

For example the macro library implementing the "gcc" ABI has the following lines:

```
unsigned _abi_list_size = 1;
AbiInfo _abi_list[1] = {{"gcc", NULL, parse_class_gcc_abi,
                         "gcc_abi_info", NULL}};
```

with the following lines in the header file:

```
module gcc_abi_info {
  macro alloc(type, size) {...}
  macro free(type, ptr) {...}
}
```

where `alloc` and `free` are called by the `new` and `delete` primitives, receptively.

The final memeber `module` is filled in by ZL when the macro library is read in, by looking for a module with the name `module_name`.

7.5.4 Mangler API

The final aspect of the ABI that ZL can control is the mangling scheme. The API to implement the alternative manglers is part of ABI switch implementation just described. The function type `MangleFun` is defined as:

```
StringObj * (*MangleFun)(Symbol *)
```

The mangler takes a symbol and transforms it into a string of the form of a pointer to `StringObj`. The string object is expected to build up a string using the `StringBuf` and then call the `freeze` method, which returns a `StringObj`. An overview of the `StringBuf` class is given in Figure 7.15.

In order to transform the string the mangler needs access to a large number of properties about the symbol. The most important of these properties is the parameter types for function symbols as they are the primary components of the mangled name. An overview of the API used for getting symbol properties is given in Figure 7.16.

Once the mangler function is defined it is necessary to register it with ABI switch. Diffrent components of the ABI may be given in diffrent libraries, and any NULL fields will simply be left alone if there where defined elsewhere for that ABI. For example, the GCC mangler is defined with the following line:

```
unsigned _abi_list_size = 1;
AbiInfo _abi_list[1] = {{"gcc", to_external_name, NULL, NULL, NULL}};
```

```
class StringBuf {
public:
    StringBuf();
    StringBuf(const char * s);
    StringBuf(const char * s, unsigned size);
    StringBuf(const StringBuf & other);
    StringBuf & operator= (const char * other);
    StringBuf & operator= (const StringBuf & sother);
    StringBuf & append(char * start, char * stop);
    StringBuf & operator+= (const char * s);
    StringBuf & operator+= (const StringBuf & s);
    StringBuf & prepend(const char * str);
    int printf(const char * format, ...);
    size_t size() const;
    bool empty() const;
    char * data();
    StringObj * freeze();
    ...;
};
```

Figure 7.15. Overview of the StringBuf class.

Type `Symbol` with **methods**:

```
const char * name()
const char * uniq_name()
Type * type()
FunType * fun_type()
Syntax * prop(UnmarkedSyntax * prop)
```

Type `Type` with **methods**:

```
Type * subtype()
int qualifiers()
bool is_scalar()
bool is_qualified()
bool is_pointer()
```

Constants:

```
TypeQualifier_CONST    = 1
TypeQualifier_VOLATILE = 2
TypeQualifier_RESTRICT = 4
```

Type `FunType` with **methods**:

```
Type * ret_type()
unsigned num_parms(const FunType *)
Type * parm_type(unsigned num)
```

Figure 7.16. Overview of the symbol API

CHAPTER 8

ZL IMPLEMENTATION DETAILS

This chapter gives the implementation details of the interesting parts of ZL.

8.1 Basic Expander and Hygiene System

This section describes the basic macro-expansion algorithm without the reparsing steps to focus on the hygiene system. For simplicity, we first assume that macro parameters and syntax forms are fully parsed; the next section gives the details.

8.1.1 The Idea

During parsing, ZL maintains an environment that maps from one type of symbol to another. Symbols in the environment's domain correspond to symbols in syntax objects, while each symbol in the environment's range is generated to represent a particular binding. Symbols in syntax objects (and hence the environment domain) have a set of marks associated with them. The set of marks are considered part of the symbol's identity. A mark is created with the `new_mark` primitive and applied to symbols during the replacement process (via `replace`). During this process, each symbol is either replaced, if it is a macro parameter, or marked. A mark also has an environment associated with it, which is the global environment at the site of the `new_mark` call.

When looking up a binding, the current environment is first checked. If a symbol with the same set of marks is not found in the current environment, then the outermost mark is stripped and the symbol is looked up in the environment associated with the stripped mark. This process continues until no more marks are left.

8.1.2 An Illustrative Example

To better understand this process, consider the code in Figure 8.1. When the first binding form "`float r = ...`" is parsed, `r` is bound to the unique symbol `$r0`, and the

```
float r = 1.61803399;

Syntax * make_golden(Syntax * syn, Environ * env) {
  Mark * mark = new_mark();
  Match * m = match_args(0, syntax (A,B,ADJ,FIX), syn);
  UnmarkedSyntax * r = syntax {
    for (;;) { float a = A, b = B;
               float ADJ = (a - r*b)/(1 + r);
               if (fabs(ADJ/(a+b)) > 0.01) FIX;
               else break; }
  };
  return replace(r, m, mark);
}
make_macro make_golden;

int main() {
  float q = 3, r = 2;
  make_golden(q, r, a, {q -= a; r += a;});
}
```

Figure 8.1. Example code to illustrate how hygiene is maintained. The `make_golden` macro will test if `A` and `B` are within 1% of the golden ratio. If not, it will execute the code in `FIX` to try to fix the ratio (where the required adjustment will be stored in `ADJ`) and then try again until the golden ratio condition is satisfied.

mapping `r => $r0` is added to the current environment. When the function `make_golden` is parsed, it is added to the environment. When the `new_mark()` primitive is parsed inside the body of the function, the current global environment is remembered. The `new_mark()` primitive does not capture local variables, since it makes little sense to use them in the result of the macro. Next, "`make_macro make_golden`" is parsed, which makes the function `make_golden` into a macro.

Now the body of `main` is parsed. A new local environment is created. When "`float q = 3, r = 2`" is parsed, two unique symbols `$q0` and `$r1` are created and corresponding mappings are added to the local environment. At this point, we have:

```
float $r0 = 1.61803399;
[make_golden => ..., r => $r0]
int main () {
  float $q0 = 3, $r1 = 2;
  [r => $r1, q => $q0, make_golden => ..., r => $r0]
  make_golden(q, r, a, {q -= a; r += a;});
}
```

The expanded output is represented in this section as pseudo-syntax that is like the input language of ZL with some additional annotations. Variables starting with `$` represent bound symbols. The `[...]` list represents the current environment in which new binding forms are added to the front of the list.

Now, `make_golden` is expanded and, in the body of `main`, we have:

```
...
[r => $r1, q => $q0, make_golden => ..., r => $r0]
for (;;) { float a'0 = q, b'0 = r;
           float a = (a'0 - r'0*b'0)/(1 + r'0);
           if (fabs(a/(a'0+b'0)) > 0.01)
             {q -= a; r += a;}
           else break; }
'0 => [r => $r0]
```

where `'0` represents a mark and `'0 => [...]` is the environment for the mark. Notice how marks keep the duplicate `a` and `r`'s in the expanded output distinct.

Now, the statement "`float a'0 = q, b'0 = r`" is compiled. Compiling the first part creates a unique symbol `$a0` and the mapping `a'0 => $a0` is added to the new environment inside the `for` loop. The variable `q` on the right-hand-side resolves to the `$q0` symbol in the local environment. A similar process is performed for the second part. We now have:

```
...
for (;;) { float $a0 = $q0, $b0 = $r1;
           [b'0 => $b0, a'0 => $a0, r => $r1,
            q => $q0, ...]
           float a = (a'0 - r'0*b'0)/(1 + r'0);
           ...}
'0 => [r => $r0]
```

Next, the statement "`float a = ...`" is compiled. A unique symbol `$a1` is created for `a` and the associated mapping is added to the local environment. Then the right-hand-side expression must be compiled. The variables `a'0` and `b'0` resolve to `$a0` and `$b0`, respectively, since they are found in the local environment. However, `r'0` is not found, so the mark `'0` is stripped, and `r` is looked up in the environment for the `'0` mark and resolves to `$r0`. We now have:

```
...
for (;;) { ...
           float $a1 = ($a0 - $r0*$b0)/(1 + $r0);
           [a => $a1, b'0 => $b0, a'0 => $a0,
            r => $r1, q => $q0, ...]
           if (fabs(a/(a'0+b'0)) > 0.01)
             {q -= a; r += a;}
           else break; }
'0 => [r => $r0]
```

Next, the `if` is compiled. The marks keep the two `a` variables in the expression `a/(a'0+b'0)` distinct, and everything correctly resolves. Thus, we finally have:

```
float $r0 = 1.61803399;
int main() {
  float $q0 = 3, $r1 = 2;
  for (;;) { float $a0 = $q0, $b0 = $r1;
             float $a1 = ($a0 - $r0*$b0)/(1 + $r0);
             if (fabs($a1/($a0+$b0)) > 0.01)
               {$q0 -= $a1; $r1 += $a1;}
             else break; }
}
```

Hence, all symbols are correctly bound and hygiene is maintained.

8.1.3 Multiple Marks

The symbols in the expansion of `make_golden` only had a single mark applied to them. However, in some cases, such as when macros expand to other macros, multiple marks are

needed. For example, multiple marks are needed in the expansion of `plus_10` in Figure 8.2. In this figure, `mk_plus_n` expands to

```
macro plus_10 (X'0) { ({int x'0 = X'0; x'0 + x;}); }
```

where the first mark `'0` is applied. A second mark is then applied in the expansion of `plus_10(x)` in `main`:

```
{ ({int x'0'1 = x; x'0'1 + x'1;}) }
```

In particular, a second mark is added to `x'0`, making it `x'0'1`. This symbol then resolves to the `x` local to the macro `plus_10`. In addition, `x'1` resolves to the global `x` constant[1] and the unmarked `x` resolves to the `x` local to `main`. Thus, hygiene is maintained in spite of three different `x`'s in the expansion.

8.1.4 Structure Fields

Normal hygiene rules will not have the desired effect when accessing fields of a structure or class. Instead of trying to look up a symbol in the current environment, we are asking to look up a symbol within a specialized subenvironment.

For example, the following code will not work with normal hygiene rules:

---

[1] In pattern based macros there is an implicit call to `new_mark` at the point where the macro was defined; hence, the `'1` mark captures the environment where `mk_plus_10` (expanded from `mk_plus_n`) is defined, which includes the global constant `x`.

```
macro mk_plus_n (NAME, N) {
  macro NAME (X) { ({int x = X; x + N;}); }
}

static const int x = 10;
mk_plus_n(plus_10, x);

int main() {
  int x = 20;
  return plus_10(x);
}
```

Figure 8.2. Example code to show how hygiene is maintained when a macro expands to another macro.

```
macro sum(q) {q.x + q.y;}
struct S {int x; int y;}
int f() {
  struct S p;
  ...
  return sum(p);
}
```

The problem is that `sum(p)` will not be able to access the fields of `p` since it will expand to "`p.x'0 + p.y'0`" with marks on `x` and `y`. The solution is to use a special lookup rule for structure fields. The rule is that if the current symbol with its sets of marks is not found in the structure, strip the outermost mark and try again, and repeat the process until no more marks are left. This process is similar to the normal lookup rule except that the subenvironment associated with the mark is ignored since it is irrelevant. In the above example, `p.x'0` in the expansion of `sum(p)` will resolve to the structure field `x` in `struct S`.

8.1.5 Replacing Context

The `get_context` and `replace_context` functions (see Section 7.3.6) can be used to bend normal hygiene rules. A *context* is simply a collection of marks. Thus `get_context` simply gets the marks associated with the syntax object, while `replace_context` replaces the marks of a syntax object. If a syntax object already has any marks associated with it, they are ignored.

8.1.6 Fluid Binding

The `fluid_binding` form (see Section 7.3.7) bends hygiene by allowing a variable to take its meaning from the use site rather than from the macros's definition site. It changes the scope of a marked variable from lexical to *fluid* and is used together with the `fluid` keyword, which temporarily binds a new symbol to the fluid variable for the current scope.

The `fluid_binding` form inserts a *fluid-binding* symbol into the environment that serves as an instruction to perform the lookup again. The symbol consists of the instruction and a unique symbol name to perform the second lookup on; the name is constructed by taking the symbol name and applying a fresh mark to it (with an empty environment). For example, "`fluid_binding this`" inserts the mapping `this => fluid(this'0)` into the

environment, where the fluid-binding symbol is represented as `fluid(SYMBOL'MARK)`. The "`fluid VAR`" form then replaces the variable `VAR` with the unique symbol name associated with the fluid binding. This has the effect of rebinding the `fluid_binding` variable to the current symbol for the current scope. For example, "`X * fluid this`" becomes "`X * this'0`" and `this'0` gets temporarily bound to the local symbol `$this0`. Finally, whenever a symbol resolves to something that is a fluid binding the symbol will be resolved again, this time using the unique symbol name in the fluid binding. For example, `this` will first resolve to `fluid(this'0)`, which then resolves to `$this0`.

To see why this method works, consider the parsing of `f`internal` from the expansion of class `C` given in Section 7.1:

```
fluid_binding this;
...
user_type C {
  ...
  macro i(:this ths = this) {(*(C *)ths)..i;}
  macro f(j, :this ths = this) {f`internal(ths, j);}
  int f`internal(C * fluid this, int j) {return i + j;}
}
```

The `fluid_binding` form (given in the prelude) is first parsed and the mapping "`this => fluid(this'0)`" is added to the environment where `'0` is an empty mark. The macros `i` and `f` in the user type `C` are also parsed and we now have:

```
user_type C {
  [f => ..., i => ..., this => fluid(this'0)]
  int f`internal(C * fluid this, int j) {return i + j;}
}
```

Now `f`internal` is parsed. Since the first parameter has the `fluid` keyword the symbol `this` is looked up in the environment and `fluid this` becomes `this'0` giving:

```
int f`internal(C * this'0, int j) {...}
```

The parameters are now parsed and added to the environment and the body of `f`internal` is expanded:

```
int f`internal(C * $this0, int $j0) {
  [j => $j0, this'0 => $this0, f => ..., i => ..., this => fluid(this'0)]
  return (*(C *)this'1)..i + j;
}
'1 => [..., this => fluid(this'0)]
```

The body of `f‘internal` is now parsed. The variable `this’1` (from the expansion of `i`) first resolves to the fluid symbol `fluid(this’0)`, which temporarily becomes `this’0` and then resolves to `$this0`. The rest of `f‘internal` is also parsed giving:

```
int f‘internal(C * $this0, int $j0) {
  return (*(C *)$this0)..i + $j0;
}
```

Hence, the `this` variable in the macro `i` gets resolved to to the `this` parameter in `f‘internal` as intended.

8.2 The Reparser

Supporting Scheme-style macros with C-like syntax turns out to be a hard problem for two reasons. The primary reason, as mentioned in Section 4.3, is that ZL does not initially know how to parse any part of the syntax involved with macros. The other and less obvious reason is that when given a syntax form such as “`syntax (x * y)`”, ZL does not know if `x` and `y` are normal variables or pattern variables until the substitution is performed. If they are normal variables, then it will be parsed as `(exp x * y)`, but if they are pattern variables, it will be parsed as `(exp (mid x) * (mid y))` where `mid` (macro identifier) is just another name for a pattern variable. ZL solves the former problem by delaying parsing as much as possible, which works nicely with ZL’s hygiene system by reducing the complexity of macro explanation from quadratic to linear. ZL solves the latter problem by installing special hooks into its Packrat parser.

8.2.1 The Idea

As already established, the `syntax ()` and `syntax {}` forms create syntax objects with raw text that cannot be parsed until ZL knows where the syntax object will ultimately be used. Thus `replace` is unable to perform any replacements. Instead, `replace` annotates the syntax object with with a set of instructions to apply later that includes two bits of information: (1) the mark to apply, and (2) the substitutions to apply.

For example, given the code:

```
int x;
Syntax * plus_x(Syntax * syn, Environ * env) {
  Match * m = match_args(0, syntax (y), syn);
  return replace(syntax (x + y), m, new_mark());
}
make_macro plus_x;
```

the call `plus_x(z)` results in `("()" "x + y"){'0; y => (parm "z")}` where the `{}` represents the annotation and `parm` is a built-in macro (see Section 7.2.3) to indicate the need to reparse. The first part of the annotation is the mark and the second is the substitution to apply. Thus the substitution is delayed until ZL knows where the call to `plus_x` will be used.

Eventually, the annotated syntax object will need to be parsed, which requires two steps. First the raw text needs to be parsed using the Packrat parser. Second the instructions in the annotations need to be applied.

Parsing the raw text creates a problem since ZL does not know which identifiers are pattern variables. Solving this problem involves a special hook into the Packrat parser, which is the purpose of the special `<<mid>>` operator shown in the grammar (Figure 7.2). The relevant bits of the grammar (with some extra required productions) are these:

```
EXP = <exp> {TOKEN}+;
TOKEN_ = <<mid PARM>> {MID} / {ID} / ...
MID = {[@$\a_][\a_\d]*} SPACING;
PARM = {STMT} EOF / {TOKEN} EOF / {EXP} EOF;
```

The `<<mid>>` operator is a special operator that matches only if the identifier being parsed is in the substitution list. When a `MID` matches, and the pattern variable is of the type that needs to be reparsed (i.e., matched with a `syntax` form), the parser adds a note as to how to reparse the macro parameter. This is either the production where it matches or the production as given in the `<<mid>>` instruction. For example, when parsing

```
("()" "x + y"){'0; y => (parm "z")}
```

as an expression, the parser is able to recognize `x` as an identifier and `y` as a `mid`. During the parsing of `x` the `MID` production is tried but it is rejected because `x` is not a pattern variable, yet when `y` is tried, it matches the `MID` production since `y` is a pattern variable. Thus the result of the parse is:

```
(exp x + (mid y PARM)){'0; y => (parm "z")}
```

After the raw text is parsed, the instructions in the annotation are applied to the subparts; if the syntax object represents raw text then the instructions are simply pushed down rather than being directly applied. In the above example this process will result in:

```
(exp'0 x'0 +'0 z)
```

That is, marks are applied and `(mid y PARM)` becomes `z`. During the substitution, the string `z` is reparsed using the `PARM` production noted in the second argument of `mid`. Hence, the string `z` becomes the identifier `z`.

The results of the reparse are then expanded and parsed as before. Marks are used as described in Section 8.1, but with the additional rule that if no marks are left and a symbol is still not found then it is assumed to be associated with a primitive form. For example, `exp'0` is assumed to represent the built in `exp` macro, since `exp` is not in the current environment. Since the result is an `exp`, it will be expanded again to become

```
(plus x'0 z)
```

which will then be converted into an AST.

8.2.2 Additional Examples

In the previous example, the result of the reparse is a fully parsed string, but this is not always the case. For example, if the macro `plus_x` were instead `plus_2x`, and the call `plus_2x(z)` expanded to:

```
("()" "2*x + y"){'0; y => (parm "z")}
```

the result will first parse to:

```
(exp ("()" "2*x") + y){'0; y => (parm "z")}
```

with `"2*x"` left unparsed. Applying the annotations will then result in:

```
(exp'0 ("()" "2*x"){'0; y => (parm "z")} + z)
```

That is, since the `"()"` syntax objects represents raw text, the instructions are pushed down on that object rather than being directly applied.

Also, in the same example, the macro parameter was just an identifier and the special `PARM` production is not needed, as it would be correctly parsed as a `TOKEN`. However, this is not always the case. For example, if the call to `plus_x` were instead `plus_x(z + 2)` the string "`z + 2`" would need to be parsed as a `PARM` since it is not a token.

8.2.3 Matching and Replacing with the `raw_syntax` Form

As the lazy substitutions of macro parameters and the reparsing are coupled, lazy substitution only applies to syntax forms that are to be reparsed, such as the `()` and `{}` forms. Syntax created with `raw_syntax` is fully parsed, and thus `replace` performs the substitutions eagerly.

8.3 Parser Details

To allow for easily adding lexical extensions, ZL uses a Packrat parser with the grammar specified as an extended PEG (see 7.2.2). When considering what parsing technology to use we also considered GLR (Generalized Left-to-right Rightmost derivation) parsing. GLR parsing differs from Packrat parsing in that the grammar is specified as a CFG (Context Free Grammar). Unlike specialized LR(k) or LL(k) parsers, a GLR parser accepts any CFG and conflicts are handled by creating multiple parse trees in the hope that the conflict will latter be resolved. Unfortunately, there is no way to know if the conflict will ultimately be resolved, as determining if a CFG is unambiguous is an undecidable problem. The worst case performance of a GLR parser is $O(n^3)$, but for most grammars the performance in practice can be made near linear. In contrast and because a PEG is a specification of how to parse the text, Packrat parsing is always unambiguous; however, the parse may not always be what was intended. In addition, Packrat parsing is guaranteed linear (although with a large constant factor) due to memorization. Packrat parsing also avoids the need for a separate lexer pass as it naturally works well with raw characters (since the PEG language is very close to the language of regular expressions used by traditional lexers). For all these reasons, and others, we chose Packrat parsing over GLR parsing.

We also chose to use Packrat parsing because the memorization can also be used to avoid quadratic parsing times with ZL's frequent reparsing of strings. For example, when parsing `(x*(y+z))` as `("()" "x*(y+z)")`, the `PAREN` production is used on `(y+z)`, since ZL must recognize the grouping. When `("()" "x*(y+z)")` is expanded, the same `PAREN` production is used. Therefore, if the memorization table for the `PAREN` production is kept after the initial parse, there will be no need to reparse `(y+z)`.

8.3.1 Performance Improvements

For ease of implementation, and unlike other Packrat parser such as Rats! [40], ZL's PEG is directly interpreted. (In other words, ZL's parser is not a parser generator.) The initial implementation of the parser was a major bottleneck. However, after making several key improvements we were able to improve the performance and memory usage of ZL by over an order of magnitude as shown in Table 8.1. The table shows numbers from a simple benchmark that consisted of compiling several nontrivial programs. These programs consisted of compiling ZL's prelude as well as several of the examples given in Chapter 5. The tests were run on an AMD Athlon(tm) 64 3000+ Processor with 1 GiB total RAM, and ZL was compiled with GCC 4.4 with basic optimization enabled.

Most of the improvements are from using better data structures. However, there were several improvements worth noting. A summary of these improvements is shown in Table 8.2.

The first improvement involved how errors are handled. Using the techniques outlined in Bryan Ford's Master's thesis [33], ZL makes a basic attempt to find the most probable reason that caused the parse to fail. This, unfortunately, involved keeping a lot of state around, which would normally not be needed. Hence, a big improvement was made by

Table 8.1. Improvements in run time and memory usage due to parser optimizations.

What	Before	After	Improvement
Avg. Run Time	1.90 sec.	0.156 sec.	12.2 times
Avg. Max Heap Usage	57.61 MiB	4.22 MiB	13.7 times

Table 8.2. Effects of individual optimizations in run time and memory usage.

Improvement	Run Time Reduction	Heap Usage Reduction
Don't Keep Error State	2.15 times	2.13 times
Keep State Between Reparses	1.21 times	1.14 times
Mark Transient Productions	1.04 times	1.68 times

simply not keeping this state around during normal parsing. If the parse failed, the text would be reparsed in a separate mode in order to find the error. This improvement led to a reduction in run-time and memory usage by a factor of around 2.1.

Another improvement worth noting was keeping the state around when reparsing strings to avoid quadratic parsing times. Unfortunately, not all productions can be kept between reparses, because sometimes the result of the parse involves a possible macro identifier (productions with the special `<<mid>>` instruction) and hence the results of the parse could change. For example, in Figure 7.2 (page 67) `TOP`, `STMT`, `EXP`, `BLOCK`, `TOKEN_`, `TOKEN`, `ID` could not be kept since they all involved a possible macro identifier. As a result of this and other factors this improvement did not have nearly as much of an effect as we had hoped, as it only lead to around a 1.2 times improvement in run-time and 1.1 times reduction in memory usage.

Finally, we implemented the ability to mark certain productions as transient (i.e., used only once) as was done in Rats! [40] to disable memoization on the production. Unlike with Rats!, however, transient productions in ZL cannot be determined statically since some productions, while appearing only once in the grammar, are in fact used more than once when reparsing. Thus, we also implemented a special profile-like mode in ZL that will output data that can be used automatically to discover transient productions and create a hint file which can then be used by ZL. In the sample grammar shown in Figure 7.2, `TOP`, `STMT`, `EXP` are all transient. In addition, `BLOCK`, `TOKEN`, `RAW_TOKEN`, and `SPACING` where also marked as transient since they are low-cost. This optimization led to a small improvement (1.04 times) in run time and a larger (1.7 times) reduction in memory usage.

CHAPTER 9

IMPLEMENTATION STATUS AND PERFORMANCE

The current ZL prototype supports most of C and an important subset of C++. For C, the only major feature not supported is bitfields, mainly because the need has not arisen. C++ is a rather complicated language, and fully implementing it correctly is beyond the scope of our research. We aim to implement enough of C++ to demonstrate our approach; in particular, we support single inheritance, but currently do not support multiple inheritance, exceptions, or templates.

As ZL is at present only a prototype compiler, the overall compile time when compared to compiling with GCC 4.4 is 2 to 3 times slower. However, ZL is designed to have little to no impact on the resulting code. ZL's macro system imposes no run-time overhead.

The ZL compiler transforms higher level ZL into a low-level S-expression-like language that can best be described as C with Scheme syntax. Syntactically, the output is very similar to fully expanded ZL as shown in Figure 4.1. The transformed code is then passed to a modified version of GCC 4.4. When pure C is passed in we are very careful to avoid any transformations that might affect performance. The class macro currently implements a C++ ABI that is comparable to a traditional ABI, and hence should have no impact on performance.

9.1 C Support

To demonstrate that ZL can support C programs, two well-known programs were compiled with ZL: bzip2 and gzip. Bzip2 was compiled without modifications, but gzip required some minor modification because it was an older C program and used some C syntax that is not a subset of C++: K&R-style function declarations were transformed into the newer ANSI C style, and one instance of `new` as a variable was renamed to `new_`.

Overall, compile times were 2 to 3 times slower with ZL in comparison to compiling with GCC 4.4. However, both programs compiled correctly, produced correct results, and had similar run times to the GCC-compiled versions.

9.2 C++ Support

To evaluate ZL's suitability to compile C++ programs, we chose to compile randprog [26], which is a small C++ program that generates random C programs. Randprog uses inheritance and other important C++ features, such as overloading and nondefault constructors. In addition, it uses a few C++ features that ZL does not yet support, so we changed randprog in small ways to compensate. These changes include reworking the command-line argument parsing, which used of a library that requires many modern C++ features; explicit instantiation of `vector` instances; changing uses of the `for_each` template function into normal `for` loops; and reworking some functions to avoid returning complex objects.

Randprog was verified to produce correct results by fixing the seed and comparing the generated program with a version of randprog compiled with GCC for several different seeds. It was also instrumented with Valgrind and found free of memory errors.

Overall compile time was around 2.5 times slower with ZL when compared to GCC 4.4. A direct run-time performance comparison is of limited usefulness, since ZL does not use the same C++ library as GCC, but the runtime performance of the ZL-compiled version of randprog was up to twice as fast as the GCC-compiled version.

9.3 Debugging Support

As ZL is only a prototype compiler only limited debugging support is provided. In particular, ZL does not provide source level debugging support. Any debugging will need to be done on the intermediate C like code. Some attempt is made to keep track of line numbers, however. These numbers can be found in the comments after each line and are only present when all code on the line ultimately comes from the same source line. Code from complicated macro transformation will not contain line numbers.

CHAPTER 10

RELATED WORK

The problem of fragile and incompatible ABIs due to software and compiler changes is well known, and there have been several attempts to address the problem. To the author's knowledge, ZL's approach of providing a small core language and letting everything else be defined as macros has not been tried before.

10.1 Binary Compatibility

The first serious attempt to solve the problem of fragile ABIs in C++ was in ΔC++ by Palay [48], but that ABI imposes a substantial performance penalty. Williams and Kindel developed a more sophisticated system with less overhead, known as the Object Binary Interface [38]. The Object Binary Interface is used only on request, and it allows for evolutionary steps, such as adding new public and protected methods and adding or removing private data members. However, it does not allow for changing the order or type of public data members; thus, it greatly reduces the problem of fragile ABIs, but does not entirely eliminate it. This ABI also imposes a higher cost when compared to the more traditional C++ ABI, and as such, is likely to affect performance, especially since all inheritance is implemented in a manner similar to how virtual inheritance is implemented in traditional C++ ABIs. Work on ΔC++ and the Object Binary Interface was done in the early 90s. Research on how to solve the problem in C++ since then is virtually nonexistent, most likely because of the inherent tradeoff between fragility and speed.

Some attempts have been made to standardize the C++ ABI between compilers for a given architecture. For example, the Itanium C++ ABI [7] aims to standardize the C++ ABI for the Itanium platform. This ABI is now used by GCC for all platforms towards the goal of providing a standard C++ ABI for GNU/Linux systems [3]. This effort has had some success, as the Intel C++ compiler also uses this ABI [6].

Since the problem of a fragile and incompatible ABIs was recognized as a serious issue that needed to be addressed, some newer languages, such as Java, specifically address the issue in the language specification. The Java concept of binary compatibility was first developed in SOM [36] and then later defined in the *Java Language Specification* (JLS) [39, 58]. In Java the ABI is completely specified in *The Java Virtual Machine Specification* [43], thus addressing the issue of incompatible ABIs.

Unfortunately supporting binary compatibility as specified in the JLS imposes a performance cost. Many Java compilers that support static compilation at first ignored binary compatibility in the interest of performance; one such compiler was the GNU Java Compiler, GCJ [4]. Later research by Yu, Shao, and Trifonov showed how to support static compilation and binary compatibility [58]. These techniques were later integrated into GCJ [53].

10.2 Scheme

ZL's design philosophy is closely related to Scheme's [51] design philosophy of providing a small core language and letting everything else be defined as macros. The hygiene and module system are similar to Chez Scheme's `syntax-case` [24] and modules [55], respectively.

While ZL's hygiene system is similar to the `syntax-case`, the data structures are different. A mark holds a lexical environment, and marks are applied during `replace` rather than to the input and result of a macro transformer. Special lookup rules search mark environments in lieu of maintaining a list of substitutions.

10.3 Other Macro Systems

There are numerous other macro systems for various languages, but apart from Scheme, few have the goal of allowing a large part of the language to be defined via macros. As such, they are either a macro system built on top of an existing language, or they lack procedural macros for general compile-time programming.

Maya [15] is a powerful macro system for Java. Maya macros (known as Mayans) support lexical extensions by extending Java's LALR(1) grammar. Like ZL's macros, Mayans are procedural and hygienic. Unlike the current version of ZL, Mayans are

modular; however, since they extend the LALR(1) grammar, conflicts may well arise when combining them. OpenJava [52] and ELIDE [20] are similar to Maya but less advanced. Neither of these systems support hygiene, and they do not support general syntax extensions.

A procedural and hygienic macro system based on the Earley [25] parser is described in Kolbly's dissertation [42]. His system is similar to Maya in that macro expansion is part of the parsing process, yet more powerful as the Earley parser can handle arbitrary grammars rather than just the LALR(1) subset. His macro system is also used in the RScheme [8] dialect of Scheme.

Fortress [10] is a new language with hygienic macro support, and the ability to extend the syntax of the language. Like ZL, it uses a Packrat parser to support lexical extensions. In addition and unlike the current version of ZL, the lexical extensions are modular and thus can be combined. Fortress macros support recursive and mutually recursive definitions unlike some other macro systems. However, macros cannot expand to other macros, and are not procedural.

The Dylan [50] language has support for hygienic macros. However, unlike ZL, one cannot really extend the grammar as macros are required to take one of three fixed forms: `def`, `stmt`, and `fun` call macros. The JSE system [13] is a version of Dylan macros adapted to Java.

MS^2 [56] is an older, more powerful macro system for C. It essentially is a Lisp `defmacro` system for C. It offers powerful macros since they are procedural, but like Lisp's `defmacro` lacks hygiene. In addition, like Dylan but unlike ZL, macros are required to take one of several fixed forms; no mechanism for general syntactic extensions is provided.

ASTEC [44] is a safer C preprocessor that is less error prone and easier to analyze. As such, it does not aim to be a complete macro system and thus has many of the same limitations of the C preprocessor in terms of power of the macro system.

The `<bigwig>` [18] language support pattern-based macros and lexical extension. However, and unlike ZL, its macros are limited in power because recursion is explicitly forbidden. By limiting the power of the macro system `<bigwig>` can support type safety and termination of the macro-expansion process.

MacroML [37] has similar aims to `<bigwig>` in that it limits what macros can do to ensure safety. While MacroML supports recursion, it does not support lexical extensions. In addition, macros are not allowed to inspect or take apart code. However, these restrictions allow macros to be statically typed. This guarantees that macro definitions are well formed and thus always produce valid code.

10.4 Ziggurat

All of the macro systems mentioned so far are either macro systems on top of a language or a macro system that is part of a language. Ziggurat [29, 28] is neither. Rather, it is a language framework centered around Scheme-like macros.

In Scheme, language extensions can be stacked to form "language towers" [30], but information on each level of the language tower is generally lost once expanded. With Ziggurat, each level in the language tower can also have static semantics. For example, fully expanded ZL code has no notion of class methods; this information is available in the higher level syntax, but is lost once classes are expanded. In Ziggurat, the higher-level class syntax is one level in the language tower, the fully expanded classes another level, and the generated C-like code another level. For analysis, each level is important; for-example, higher-level class information (and, in particular, a notion of class methods) is needed in order to effectively perform devirtualization. With a Ziggurat style type abstraction in ZL, it may even be possible to avoid having a type system, and hence user types, built into the language as they could simply be static semantics of one of the language layers.

Ziggurat, however, only works in S-expression syntax. It has a language layer for C, but only after it is converted to S-expression syntax. Thus just as ZL can benefit from Ziggurat type abstraction, Ziggurat can benefit from having a more sophisticated parsing layer, perhaps one similar to ZL (after it has suitably been extended to support more module syntax) that converts higher level syntax to S-expression syntax.

10.5 Extensible Compilers

Macros are one approach to providing an extensible compiler, but a more traditional approach is to provide an API to directly manipulate the compiler's internals, such as the AST. On the surface this approach may seem more powerful than a macro system, but we

believe a macro system can be equally powerful with the right hooks into the compiler.

Macros also elevate the extensions to the level of a library. For example in ZL being able to define classes with different ABIs arises naturally from the macro based approach to defining classes. With a traditional extensible compiler, such a feat will require extra book-keeping. Nevertheless, some tasks involving complicated program transformations such as optimizations are better served by the more traditional extensible compiler approach.

Xoc [21] is an extensible compiler that supports grammar extensions by using GLR (Generalized Left-to-right Rightmost derivation) parsing techniques. Xoc's primary focus is on implementing new features via many little extensions, otherwise known as plugins. This approach has an advantage over most other extensible compilers in that the extensions to be loaded can be tailored for each source file. As such, Xoc provides functionality similar to that of traditional macro systems.

METABORG [19] is a method for embedding domain-specific languages in a host language. It does this by transforming the embedded language to the host language using the Stratego/XT [54] toolset. Stratego/XT supports grammar modifications using GLR parsing techniques.

Polyglot [46] is a compiler front-end framework for building Java language extensions; however, since it uses an LALR parser, extensions do not compose well. JTS [17] is a framework for writing Java preprocessor with the focus on creating domain-specific languages. Stratego/XT [54] is a compiler framework whose primary focus is on stand-alone program transformation systems; it also supports grammar modifications using GLR parsing techniques. CIL [45] focus in on C program analysis and transformation, and as such, does not support grammar modifications. Again, as external tools, these systems all represent an approach different from ZL's support for extension within the language.

CHAPTER 11

DISCUSSION AND FUTURE WORK

The main use of ZL in this dissertation has been to mitigate ABI compatibility problems. In the process, we have also demonstrated how a powerfull Scheme-like macro system can be used to build C++ constructs over a C-like core. As such, ZL has many other potential uses beyond addressing ABI compatibility problems. In this chapter, we will evaluate ABI problems solved, outline additional work to be done, explore other uses of ZL, and outline areas of future research.

11.1 Evaluation of ABI Problems Solved

In Chapter 5 and 6 of this dissertation, we have solved many of the ABI problems outlined in Chapter 3. In addition, we outlined what needed to be done to solve the remainder of those problems in Section 5.7. We have demonstrated that we can use several ABIs at once in addition to using ZL to help maintain binary comparably while still matching another compiler's ABI (GCC's).

Matching other compilers' ABIs would contribute little towards supporting the thesis, but would improve the utility of ZL. Other than GCC, the main target ABI would be the Visual C++ ABI. Work involved towards matching this ABI includes 1) porting ZL to Windows, 2) writing a new mangler function, 3) adding support the "this" calling conversion in the ZL backend. The version of GCC that the ZL backend was created from does not support the "this" calling conversion, yet newer versions do, so implementing (3) involves either back-porting the changes in GCC that support the new calling conversion or forward-porting the front-end changes to the latest version of GCC. It is important to note that class layout for ZL ABI, GCC ABI, and Visual C++ ABI are the same except for perhaps the issue of multiple destructors; thus there is very little work to be done in that area.

11.2 Error Messages and Debugging Support

ZL attempts to provide helpful error information when expanded code has syntax or related errors by to providing a backtrace of what is expanded from where, but the backtrace is inexact. In addition, if the code was preprocessed using the C preprocessor source locations will correspond to the preprocessed output rather than the original source code.

As mentioned in Section 9.3, ZL does not provide any source-level debug support, so any debugging is done on the generated code. In some cases, line numbers are provided in comments, which can help, but the user must still look at two source files to find the problem. Worse, as is the case with error messages, if the code needed to first go through the C preprocessor, these line numbers correspond to the output of the prepossessed code, rather than the original source. Needless to say, this is unacceptable for compiling all but the most trivial of programs, and in fact has made developing the class macros more difficult than it could be.

To improve both situations, several things need to be done. The rest of this section details some of those steps.

11.2.1 Handing of Code Needing the C Preprocessor

To provide better debugging and error message support, the C preprocessor needs to be integrated into the ZL parser, or at very least ZL needs to respect the `#line` directives generated by the external preprocessor. Integrating the C preprocessor will in principle allow keeping track of the exact location of each character in the input; in reality, this support will also require changes in how ZL maintains source locations, as ZL can only assign one source location to each token, but with preprocessed code the same token can have multiple source locations.

Even with the limitations of one source location per token, an integrated C preprocessor is needed to have any hope of keeping track of source code at the character level. One obvious example where character level information can get lost is when part of the line contains an expansion of a preprocessor macro. In fact, the problem is worse than that. The C preprocessor makes no guarantee to preserve the amount of whitespace in a line. Thus, even without any transformations, code such as

```
int x  =  y + z;
```

could become:

```
int x = y + z;
```

so that nearly any character level information on preprocessed output will be inexact.

However, an integrated preprocessor is not required in order to keep track of source code location at the line level as long as ZL respects the `#line` directive. The main barrier to supporting `#line` is that, as ZL normally keeps track of source code location at the character level, error messages tend to rely on this information, and can thus be misleading. The easiest solution to this problem is to just ignore it, and accept the fact that character location within a line may be wrong. A slightly better solution is to maintain a flag to indicate that character location may be wrong and use other means to help the user determine the location within the line that the error is located (such as giving a code snippet), which is what most C compilers that do not have an integrated C preprocessor currently do.

11.2.2 Source Level Debugging

Once correct handling of preprocessed code is done, the next step is to get source level information into the object file as debugging symbols. Adding support for line numbers is simply a matter of figuring out how to get ZL's back-end compiler (the modified version of GCC) to generate said symbols. Unfortunately, since the back-end compiler is really just a modification of the C front-end, this is a more difficult task than it should be as the C front-end provides no easy means of setting line numbers directly.

In addition to providing line number information, good source level debugging needs the ability to examine the value of variables. The variable names used in the generated source code have some correspondence to the original variable names, but in most cases they are mangled to avoid any possibility of conflict. For example, a local variable `x` will get renamed to `x$1`, which can be confusing to the end-user that should not need to worry about the implementation details. One solution to this problem is to only rename variables when necessary (for example, to prevent conflicts in macro expanded code), and another is to figure out how to get GCC to give alternative names (i.e., the original name as it appears in the source code) in the debugging output. Neither solution is simple to implement.

Even when debugging with the original names, there is still the problem of how to refer to variables introduced in the macro that conflict with local names, such as in the expansion of the `or` macro in Section 4.2 (page 27). Proper support for macro-introduced variables will likely require additional support from the debugger. Without the debugger support, the best solution is to use a consistent naming scheme for marked (i.e., macro-introduced) variables.

11.2.3 Better Support for Macro Expanded Code

In addition to adding support for referring to macro-introduced variables, another challenge in debugging macro expanded code is the fact that the same piece of code can have multiple locations: the location of the macro call site, and the location within the macro definition. Fortunately, this situation is similar to the case of inline code so there is already some debugging support. The main issue is generating the correct symbols.

Before the symbols (that contain the location for both the call site and definition site) can be generated, ZL needs to able to know what the locations are. As already mentioned, ZL does make some attempt to provide this via a backtrace of what was expanded from where, but it does not always do a good job. Section 11.5.1 discusses improvements on how macro expansion is handled internally that will likely greatly improve the quality of the backtrace.

11.3 C++ Template Support

On the surface, a good macro system can replace the need for C++ templates as they are similar to macros. However, templates provide at least one functionally that normal macro systems do not provide: the automatic installation of template instances as needed. For example, given

```
template<class T> class vector {...};

int foo() {vector<int> vec1;}
int bar() {vector<int> vec2;}
```

an instance of `vector<int>` needs to be generated for `foo()` and `bar()`. Furthermore, both functions need to use the same instance, which is important if the class has any static data members.

One approach that almost works is to have both the template definition and the use be macros. Then, `vector<int>` will be a macro call that can generate the necessary code. The problem is that the code generated will not be in the correct scope; it will be local to, say, `foo()` when it needs to be in global scope. Thus, at bare minimum some support will be needed to eject new symbols in the global environment while inside a function. There is also the problem of having multiple instances of `vector<int>`, but some bookkeeping within the macro can avoid that problem.

With explicit installation ZL's macro system can provide basic template functionally. In fact, ZL provides a macro to create instances of the `vector` template class.

While ZL's macros can provide basic template support, a correct implementation of C++ templates is more involved due to, in part, many difference in how ZL macros and templates behave; for example, the scoping rules are different. In addition, C++ templates provide advanced features, such as partial specialization and function templates, that may be hard to provide via a pure macro system. Function templates are especially tricky because special syntax is not used when calling a function template, instead the instantiation is part of the normal overloading rules. For example, given

```
template<class T> void f(T) {...}
void f(int) {...}
void f(void *) {...}
```

the call `f(6.8)` will instantiate the template version (with `double` as the `T` parameter) and `f(6)` will call the nontemplate version (the second one, with the `int` parameter). The rules are also tricky; for example, the call `f(&x)` (that is pass in a pointer to `x`) will instantiate a new version of `f` rather than calling the nontemplate version of f with the `void *` parameter, as might be expected since any pointer type can be implicitly converted to a `void *`.

11.4 C++ Support in General

As discussed in Section 9.2, ZL only supports a small subset of C++. The work needed to support templates was discussed in the last section. Here is a brief run-down of additional C++ features not implemented and the work involved to implement them:

- *Namespaces.* More Code.

- *RTTI.* More Code.
- *Multiple Inheritance.* The type system and class macro will likely need to be completely rewritten.
- *Exceptions.* Exceptions are a very low-level feature, which a compiler to a C-like language can not implement effectively. In addition it will be impossible to match any compiler's ABI at the C level. Nevertheless, there is built in exception support in GCC backend so it may be possible to tap into that code to at least implement the GCC ABI.

11.5 Enhancements to ZL's Macro System

ZL provides a powerful, hygienic macro system. The choice of using an intermediate s-expression for all macro expansion simplified the design of ZL. In addition, and although not directly related to the macro system, the choice of using an intermediate s-expression language allowed for easily adding rarely used primitives without having to define higher-level syntax for it.

Nevertheless, there are a number of improvements that can be made to the macro system. This section explores many of them.

11.5.1 Always Reparsing

As mentioned in Section 8.2.3, ZL's macro expander is a mixture of eager and lazy expansion. Expansion is eager when the `raw_syntax` form is used and lazy otherwise. As is well known [24], eager expansion can lead to quadratic expansion time. More, importantly the mixture greatly complicated the process of tracking what was expanded from where, which makes providing meaningful backtraces difficult (as mentioned in Section 11.2).

Thus, it makes sense to make all expansion lazy. In ZL lazy expansion is coupled with the reparsing processes; thus the idea is to always reparse no matter what form is used to create syntax. In particular, in order to make all expansion lazy the `raw_syntax` form will need to be handled in the same way that `syntax` forms are currently handled (see Section

8.2.1) except that, when it comes time to parse the string, the s-expression parser is used rather than the PEG parser.

An important side effect of the change (to always reparsing) is that pattern variables will get recognized like they are with the Packrat parser, and thus get transformed from bare identifiers to `mid`'s; for example, if `X` was a pattern variable then it would get transformed to `(mid X)`. In addition to eliminating some special cases, this change will also avoid confusing single letter pattern variable with "quote" syntax forms, such as `(s STRING)`. Before, the use of single letter pattern variables as the first part of a syntax object was pragmatic because the template `(s n)` will always represent the string `n` regardless of if `s` is a pattern variable. With the addition of the reparser for s-expression forms, `(s n)` will become `((mid s) (mid n))` if `s` is a pattern variable and the syntax was created using `raw_syntax`.

11.5.2 Matching Literals Hygienically

ZL provides limited support for matching literals; for example, Figure 5.1 (page 35) uses `(pattern ({...} @body))` to match against the `{...}` literal. However, literals are matched symbolically with no regard to lexical context. Even outside of `(pattern ...)`, the current ZL implementation completely ignores hygiene when matching keyword like symbols.

This limitation is not a fundamental flaw, however, and it can be fixed using ZL's hygiene model. The basic idea to implementing hygienic keyword matching is to make the keyword part of the environment and use an extended version of `match` that takes into account both the environment in which the macro was called and the environment in which the macro was defined. In this model, the `keyword_binding` form would introduce a keyword into the environment; for example,

```
keyword_binding {...};
```

would make the `{...}` literal part of the environment. The extended version of `match` will then take two extra parameters: the environment passed into the macro, and a mark (which holds the environment in which the macro was defined). For example,

```
Syntax * parse_myclass(Syntax * syn, Environ * env) {
  Mark * mark = new_mark();
  Match * m = match
      (0, raw_syntax(_ name @ (pattern ({...} @body))), syn, env, mark);
  // ...
}
```

will hygienically match the `{...}` literal. The extended version of `match` will first look up the literal `{...}` in the environment associated with the mark (i.e., the environment where the macro was defined). When `match` finds the literal `{...}` in the syntax object `syn`, it will look up the symbol again, but this time in environment stored in `env` (i.e., the environment where the macro was called), and the pattern will only match if both `{...}` resolve to the same symbol.

11.5.3 Using Marks for Inner Namespaces

Although not strictly related to the the macro expander, ZL's hygiene model can eliminate the need for a separate data structure to represent inner namespaces (see Section 7.1, page 63) as the role an inner namespace provides is very similar to the role marks provide.

Instead of inner namespaces, a special mark will be used known as a "namespace" mark that has that has the following properties:

- The mark is never stripped.
- The mark is always applied before other marks.
- If multiple namespace marks are applied then sort them; the exact ordering is unimportant as long as it is stable within the compilation unit.

The syntax used to create namespace marks is unchanged. That is, namespace marks can be created with the syntax:

```
make_inner_ns NAMESPACE
```

and the syntax `'NAMESPACE` syntax will apply a namespace mark to a symbol.

11.6 Support for an Extensible Parser

Currently, lexical extensions involve modifying the grammar specification. It would be better if this was modular so that using techniques such as `fix_size` will not require

modifying the core grammar. Much work in the area of module extensions to a PEG grammar has been done in Rats! [40] system. ZL can use many of these techniques to provide for modular lexical extensions.

11.7 Beyond ABI Compatibility

ZL has many potential uses beyond addressing ABI compatibility issues. This section explores some of them.

11.7.1 Type Safe and Extensible `printf`

C's `printf` function is very powerful, yet it is also dangerous as it is a variadic function that is not type safe. That is, the following is allowed

```
int x;
printf("%f", x); // %f expects a double
```

and will not create either a compile or run-time error. Rather the code will lead to undefined behavior by the C standard. This problem is a common enough that the GCC compiler provides an extension (via the `format` attribute) to warn about such type mismatches. Even with the extension, `printf` is not extensible, in that it can not be adopted to support user-defined types.

C++ deals with the problem by providing an alternative mechanism for formatting output by overloading the the << operator. This mechanism, while type-safe and extensible, is not nearly as convenient to use as specifying a `printf`-style template.

With ZL it is possible to have the best of both worlds—to specify formatting via a template but also make the call type safe. To start with, a simple `printf` macro could simply implement GCC's `format` attribute. The macro can parse the format string and then check the type of the parameters to make sure they are compatible, then ultimately pass the result to the `printf` function. A more involved macro will allow extending the format string to support user types. A simple implementation could convert the user type to a string and pass that string onto `printf`. A more involved implementation could work more like C++ I/O but with the benefit of retaining the syntax of C's `printf`.

The Boost Format library [9] provides `printf` style formatting support for C++, which works via clever use of operator overloading. However, since C++ provides no way to parse

the format string at compile time, any type mismatches are detected at run time rather than at compile time, which means additional overhead that a macro system can avoid.

A type-safe `printf` was explored in the context of Scheme in an earlier paper by Herman and Meunier [41]. This paper also used macros to embed SQL and to compile regular expressions. Section 11.7.3 discusses how ZL can be used to embed SQL. Compiling regular expression with ZL is also possible but not discussed in this dissertation.

11.7.2 Variable Interpolation

Many dynamic languages provide support for interpolation variables in strings; for example, in such languages, the string `"Total: $total"` interpolates the local variable `total` into the string. Variable interpolation at times is more convenient than using a format string. With most dynamic languages, using a variable that is not in the current scope will only result in a error once the string is used.

C and C++ lack this feature, but with ZL it can easily be added via a macro. The macro can parse the string and transform the code to how it would be written without variable interpolation support. The macro has the added benefit that using a variable that is not in the current scope will result in a compile-time error, rather than a run-time error.

11.7.3 Embedding SQL

ZL's parsing strategy was deliberately chosen to make embedding languages within each other easy without having to make extensive grammar modifications and hence minimize the possibility of conflicts. For example, the SQL language can be embedded into ZL by adding a single syntactic form:

```
sql (...)
```

Anything between the `()` is parsed using an alternative SQL parser. The only requirement is that the SQL code does not introduce any unbalanced grouping characters.

Once embeded, the SQL syntax can access local variables and can even create new ones. For example, to insert a row into the table using the local variables `name` and `address` one could simply use

```
sql( INSERT INTO addresses (name, address) VALUES (name, address) );
```

and the SQL parser will know that the first `(name, address)` are not part of an expression but rather field names, while the second instance contains expressions that can refer to local variables. In addition, the parser will interpolate the values in a safe manner to avoid SQL injection attacks.

SQL code can also introduce local variables; for example, the code

```
foreach (sql(SELECT name, address FROM addresses)) {
  ...
}
```

creates two lexically scoped variables, `name` and `address`, that can be accessed inside the for loop. If it would be more convenient to access the rows outside of a loop, a structure can be created where each struct field corresponds to a field of the result; for example, in the above query the struct will have two fields: `name` and `address`.

11.8 Areas of Future Research

One area that needs to be explored is how to provide better source level debugging in the presence of macros, in addition to providing useful error messages. Some work has been done on this problem with Scheme macros [22], but ZL macro expander is different enough to pose its own unique challenges.

ZL's hygiene implementation is similar to Scheme's [24], but it is not the same. Future work in this area is a more formal description of the hygiene system and to determine in what ways it differs from Scheme's.

11.9 Alternative Research Direction

This dissertation solved the ABI problem through a macro system. An alternative approach is to develop a *model specification language* such as was done with calling conventions by Bailey and Davidson [14] via CCL, and to some extent Olinksy, Lindig, and Ramsey [47] via staged allocation. A model specification language is a precise description of how to implement C++ constructs.

The model specification language for calling conventions was developed by observing precisely the steps involved in making a call across a wide variety of architectures and generalizing it. A language for the the more general C++ ABI will involve a similar

process. In addition, different languages will likely need to be developed for different aspects of the ABI as they involve different concerns; for example, the specification for calling conventions is mostly about the placement of parameters in registers and on the stack, with inheritance the specification is mostly about data layout, and with exceptions it is mostly about stack-unwinding.

A good starting place would be to formalize the description from an existing ABI specifications (such as the Itanium C++ ABI) then try to adopt the language developed to a different ABI. Adapting it to the Visual C++ ABI will likely require some degree of reverse-engineering since many parts of the ABI are undocumented.

All in all, developing the specification is a topic for another dissertation, due to the amount of work involved and the fact that each aspect of the ABI will need to be handled separately.

CHAPTER 12

CONCLUSION

Binary compatibility is a serious problem for software evolution in C++. C++ ABIs tend to be fragile because they are optimized for speed rather than robustness. Thus, library implementers have developed a number of programming idioms to help mitigate the problem. Due to the sheer number of idioms and the trade-offs involved, adding them as language extensions is infeasible. In addition, C++ ABIs differ between compilers, and hence, switching compilers often breaks binary compatibility.

We have solved this problem using ZL. ZL is a C++-compatible language in which high-level constructs, such as classes, are defined using macros over a C-like core language. ZL solves the problem of binary compatibility by using macros to automate the use of programming idioms that programmers would use to mitigate the problem. When programming idioms are not sufficient, ZL gives the programmer complete control over the ABI by providing a customizable class macro. The ZL macro system benefits library implementers and consumers who do not need to know the full details of how macros work, as library implementers can just use the macro libraries written by the tool implementers. At the same time, ZL makes the job of tool implementers easier when compared to a traditional compiler system.

For the current implementation of ZL, see the ZL web page available at `http://www.cs.utah.edu/~kevina/zl/`.

APPENDIX

OVERHEAD OF THE PIMPL IDIOM

The pimpl idiom (see Section 3.2.2) adds a small amount of overhead. How significant this overhead is depends on a large number of factors. To get an idea of this overhead we measured the performance of a program that performs a simple calculation on a large number of objects, both with and without the pimpl idiom.

Figure A.1 and A.2 show the class used for the tests. The class contains eight integer data members and is designed to mimic the size of a mildly complex class used in practice. Figure A.3 shows the code used in the test. The test is broken down in three phases: the first phase *initializes* the objects; the second phase *computes* using the objects; and the final phase *destroys* the objects. Not included in any of the phases is code to allocate and free the memory used for the objects.

All of the objects used in the test are allocated in a continuous region of memory. The constant `COUNT` is the number of objects to allocate. After the objects are allocated and initialized, `compute` is called `LOOP` number of times. The `compute` function iterates through the objects 4 times, each time performing a summation over some of the fields. The main point of the calculation is to access the data members. The actual result of the calculation is immaterial; a simple summation was chosen to measure the cost of accessing the fields and not the calculation itself. The constant `A_PRIME` can be set to a prime number to access the objects in a nonlinear fashion (the default value is `1`).

Three tests are performed. In the first test, "In Cache", `COUNT` is set to 1 Ki and `LOOPS` is set to 64 Ki. The idea of this test is to measure the overhead of the pimpl idiom when all the objects can fit in the cache (likely the L1 cache). In the second test, "Predictable", `COUNT` is set to 8 Mi and `LOOPS` is set to 8. The idea of the second test is to measure the overhead when all the objects can not fit in any cache (the total size of the objects on a typical 32-bit architecture is 256 MiB) but are accessed in a predicable way. In the

```
class Test {
private:
  unsigned _a;
  ...
  unsigned _h;
public:
  Test(unsigned x) : _a(x), ... _h(x+7) {}
  unsigned a() {return _a;}
  ...
  unsigned h() {return _h;}
};
```

Figure A.1. Class used in test.

```
class TestImpl {
  unsigned _a;
  ...
  unsigned _h;
public:
  Test(unsigned x) : _a(x), ... _h(x+7) {}
};

class Test {
  TestImpl * impl;
public:
  Test(unsigned x) {impl = new TestImpl(x);}
  unsigned a() {return impl->_a;}
  ...
  unsigned h() {return impl->_h;}
  ~Test() {delete impl;}
};
```

Figure A.2. Same class (Figure A.1) but refactord to use the pimpl idiom.

```
unsigned res = 0;

int main() {
  Test * objs = (Test *)malloc(sizeof(Test) * COUNT);
  init(objs);
  for (unsigned i = 0; i != LOOPS; ++i)
    res += compute(objs);
  destroy(objs);
  free(objs);
}

void init(Test * objs) {...}

unsigned compute(Test * objs) {
  unsigned total = 0;
  unsigned i;

  i = 0;
  do { total += objs[i].a() + objs[i].b() + objs[i].c() + objs[i].d();
       i = (i + A_PRIME) % COUNT;
  } while (i != 0);

  do { total += objs[i].e() + objs[i].f();
       i = (i + A_PRIME) % COUNT;
  } while (i != 0);

  do { total += objs[i].g();
       i = (i + A_PRIME) % COUNT;
  } while (i != 0);

  do { total += objs[i].h();
       i = (i + A_PRIME) % COUNT;
  } while (i != 0);

  return total;
}

void destroy(Test * objs) {...}
```

Figure A.3. Simplified version of code used to test the overhead of the pimpl idiom.

final test, "Unpredictable", `COUNT` and `LOOPS` are set as before and, in addition, `A_PRIME` is set to a large prime (369983). The idea of the final test is to access the objects in a unpredictable order to guarantee cache-misses nearly every time. In all tests, the total number of operations performed by the compute phase is the same.

Table A.1 shows the results of the running the tests. Each test was run 60 times and the average times were taken. The test were performed on a 3.20 GHz Intel Core i3 Processor (with two cores) and compiled with a 32-bit version of GCC 4.4.3 on Ubuntu 10.04 using the `-O3` option. In the table, "overhead" is the extra time spent performing the test, which is not part of one of the three phases. The cost of the overhead phase consists primarily of the cost to allocate and free the memory for the objects. Note that when the pimpl idiom is used, the cost of allocating the `impl` object is included as part of the first phase, rather than as part of the overhead.

Depending on how the objects are accessed the slowdown in the main phase (compute) varies from 1.0 to 1.7. When the working-set size is small enough that everything fits in the

Table A.1. Overhead on using the pimpl idiom.

Test	Phase	Normal (cpu time in seconds)	Pimpl (cpu time in seconds)	Slowdown (factor of)
In Cache	Initialize	0.000 ± 0.000	0.000 ± 0.000	-
	Compute	0.335 ± 0.007	0.336 ± 0.005	1.00
	Destroy	0.000 ± 0.000	0.000 ± 0.000	-
	Overhead	0.000 ± 0.000	0.000 ± 0.000	-
	Overall	0.335 ± 0.007	0.336 ± 0.005	1.00
Predictable	Initialize	0.112 ± 0.006	0.413 ± 0.010	3.69
	Compute	1.051 ± 0.011	1.375 ± 0.015	1.31
	Destroy	0.000 ± 0.000	0.155 ± 0.008	∞
	Overhead	0.015 ± 0.005	0.004 ± 0.005	0.23
	Overall	1.178 ± 0.012	1.947 ± 0.022	1.65
Unpredictable	Initialize	0.113 ± 0.005	0.413 ± 0.015	3.65
	Compute	7.085 ± 0.057	12.087 ± 0.084	1.71
	Destroy	0.000 ± 0.000	0.153 ± 0.009	∞
	Overhead	0.015 ± 0.005	0.003 ± 0.005	0.24
	Overall	7.213 ± 0.058	12.656 ± 0.093	1.76

cache (as in the "In Cache" test), the pimpl idiom has no measurable overhead. Things are more interesting when the working-set is too large to fit in any cache. When the objects are accessed in a predicable linear fashion (as in the "Predictable" test), the pimpl idiom has an overhead of around 31%. However, when the objects are accessed in a unpredictable way, the overhead is 71%.

The main cost of the pimpl idiom is not the extra indirection but the extra cost of allocating and freeing the `impl` object. The extra cost of allocation (the initialize phase) causes a slowdown of factor of around 3.7 in our tests. The pimpl idiom also has a cost when the object is freed; it adds 0.15 seconds to the overall runtime.

For the "In Cache" test, not enough objects were allocated to measure anything and hence the cost is 0.000 for all but the compute phase. For the other tests, the already small run-time in the overhead phase is smaller when the pimpl idiom is used, due to the fact the the actual object is smaller (it shrinks from the size of 8 integers to the size of one pointer) and there is hence less memory to allocate and free.

Overall, depending on how the objects are accessed, the overhead of using the pimpl idiom is between 0 and 76%. Our tests aim for the worst case scenario where the objects are not in the cache and the computation performed on the data members is trivial. In addition, all objects were allocated at once in a single block of memory, and the ratio between the number of operations performed on the object and the number of objects allocated is low. In real programs, objects are likely to be allocated with `new`, and thus the slowdown due to the pimpl idiom is likely to be less. In addition, real programs are likely to perform far more operations on the allocated objects, and thus marginalize the cost of the initialize phase. For all these reasons, the 76% overhead is likely to be a upper bound to the performance impact in real programs, and the actual value is likely to be closer to the lower bound of 0% than to the upper bound.

REFERENCES

[1] Aspell C API reference. http://aspell.net/man-html/Through-the-C-API.html.

[2] Aspell web site. http://aspell.net.

[3] A common C++ ABI for GNU/Linux. http://gcc.gnu.org/gcc-3.2/c++-abi.html.

[4] GCJ web site. http://gcc.gnu.org/java/.

[5] GNOME. http://www.gnome.org.

[6] Intel C++ compiler man page. Available at http://software.intel.com/en-us/intel-compilers/.

[7] Itanium C++ ABI (revision: 1.86). http://www.codesourcery.com/cxx-abi/abi.html.

[8] RScheme web site. http://www.rscheme.org/rs/.

[9] The Boost Format library. http://www.boost.org/doc/libs/release/libs/format/doc/format.html.

[10] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proc. Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

[11] Kevin Atkinson and Matthew Flatt. Adapting Scheme-like macros to a C-like language. In *Proc. Workshop on Scheme and Functional Programming*, Portland, Oregon, 2011.

[12] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. ABI compatibility through a customizable language. In *Proc. Generative Programming and Component Engineering (GPCE)*, pages 147–156, Eindhoven, The Netherlands, 2010.

[13] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proc. OOPSLA*, pages 31–42, Tampa Bay, FL, 2001.

[14] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Proc. POPL*, pages 298–310, 1995.

[15] Jason Baker and Wilson C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. PLDI*, pages 270–281, Berlin, Germany, 2002.

[16] Eli Barzilay, Ryan Culpepper, and Matthew Flatt. Keeping it clean with syntax parameters. In *Proc. Workshop on Scheme and Functional Programming*, Portland, OR, 2011.

[17] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proc. Intl. Conf. Software Reuse (ICSR)*, page 143, 1998.

[18] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 31–40, Portland, OR, 2002.

[19] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. OOPSLA*, pages 365–383, Vancouver, BC, Canada, 2004.

[20] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proc. Conf. Aspect-Oriented Software Development (AOSD)*, pages 10–18, Enschede, The Netherlands, 2002.

[21] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 244–254, Seattle, WA, 2008.

[22] Ryan Culpepper and Matthias Felleisen. Debugging hygienic macros. *Science of Computer Programming*, July 2010.

[23] R. Kent Dybvig. Syntactic abstraction: the syntax-case expander. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 25, pages 407–428. O'Reilly and Associates, June 2007.

[24] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[25] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[26] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proc. Intl. Conf. on Embedded Software (EMSOFT)*, 2008.

[27] Matthias Ettrich and Lubos Lunak. KDE developer's corner: Binary compatibility issues with C++. `http://developer.kde.org/documentation/other/binarycompatibility.html`.

[28] David Fisher. *Static Semantics for Syntax Objects*. PhD thesis, Northeastern University, 2010.

[29] David Fisher and Olin Shivers. Building language towers with Ziggurat. *Journal of Funtional Programming*, 18(5–6):707–780, September 2008.

[30] Matthew Flatt. Composable and compilable macros: You want it when? In *Proc. Intl. Conf. Functional Programming (ICFP)*, pages 72–83, Pittsburgh, Pennsylvania, 2002.

[31] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[32] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, 1963.

[33] Bryan Ford. *Packrat Parsing: A Practical Linear-Time Algorithm with Backtracking*. Master's thesis, Massachusetts Institute of Technology, 2002.

[34] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proc. Intl. Conf. Functional Programming (ICFP)*, pages 36–47, Pittsburgh, PA, 2002.

[35] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proc. POPL*, pages 111–122, Venice, Italy, 2004.

[36] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *Proc. OOPSLA*, pages 426–438, Austin, TX, 1995. ACM Press.

[37] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *Proc. Intl. Conf. Functional Programming (ICFP)*, pages 74–85, Florence, Italy, 2001.

[38] Theodore C. Goldstein and Alan D. Sloane. The object binary interface: C++ objects for evolvable shared class libraries. In *Proc. USENIX C++ Technical Conf.*, pages 1–20, 1994.

[39] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2005.

[40] Robert Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51, Ottawa, Ontario, 2006.

[41] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proc. Intl. Conf. Functional Programming (ICFP)*, pages 16–27, Snow Bird, UT, 2004.

[42] Donovan Kolbly. *Extensible Language Implementation*. PhD thesis, Univ. of Texas, Austin, 2002.

[43] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[44] Bill McCloskey and Eric Brewer. ASTEC: a new approach to refactoring C. In *Proc. ESEC/FSE-13*, pages 21–30, Lisbon, Portugal, 2005.

[45] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. Conf. Compiler Construction*, pages 213–228, 2002.

[46] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. Conf. Compiler Construction*, pages 138–152, 2003.

[47] Reuben Olinksy, Christian Lindig, and Norman Ramsey. Staged allocation: A compositional technique for specifying and implementing procedure calling conventions. In *Proc. POPL*, pages 409–421, Charleston, SC, January 2006.

[48] Andrew Palay. C++ in a changing environment. In *Proc. USENIX C++ Technical Conf.*, pages 195–206, 1992.

[49] Peter Potrebic. What's the Fragile Base Class (FBC) Problem? *BE NEWSLETTER – The Newsletter for BeOS Developers and Customers*, June 25 1997.

[50] Andrew Shalit, David Moon, and Orca Starbuck. *Dylan Reference Manual*. Addison-Wesley, 1996.

[51] Michael Sperber (Ed.). *The Revised^6 Report on the Algorithmic Language Scheme*. Cambridge University Press, 2007.

[52] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proc. 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000.

[53] Tom Tromey and Andrew Haley. GCJ: The new ABI and its implications. In *Proc. GCC Developers' Summit*, pages 169–174, 2004.

[54] Eelco Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. In Lengauer et al., editor, *Domain-Specific Program Generation*, Lecture Notes in Computer Science, pages 216–238. Spinger-Verlag, June 2004.

[55] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proc. POPL*, pages 203–215, San Antonio, TX, 1999.

[56] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. PLDI*, pages 156–165, Albuquerque, NM, 1993.

[57] Sara Willliams and Charlie Kindel. The component object model: A technical overview. *Dr. Dobbs Journal*, December 1994. Also at `http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp?frame=false`.

[58] D. Yu, Z. Shao, and V. Trifonov. Supporting binary compatibility with static compilation. In *Proc. Java Virtual Machine Research and Technology Symposium (JVM)*, pages 165–180, 2002.