Domain Specific Language Extensions

Kevin Backhouse

March 22, 2000

Acknowledgements

I would like to thank Microsoft Research for its generous financial support. I would also like to thank Bernard Sufrin for his guidance. In particular, I would like to thank him for suggesting the case study in this dissertation. Although it took some time, it was a very worthwhile exercise! Finally, many thanks to Eric Van Wyk who has spent a lot of time reading and correcting drafts of this document.

Contents

1 Introduction **6**

1.1 A Categorisation of Programming Languages 7

1.2 Pros and Cons of Domain Specific and Domain Adapted Languages 8

1.3 Domain Specific Language Extensions 9

1.4 Intentional Programming 9

1.5 Requirements of an Intentional Programming Environment . . 10

1.5.1 Ecology 11

1.6 Previous Work 12

1.6.1 Macros 12

1.6.2 Extensible Compilers 13

2 GUI Language Extensions **15**

2.1 The Microsoft Foundation Classes 15

2.1.1 The MFC Application Framework 16

2.1.2 The Document-View Architecture 17

2.1.3 The Message Mapping System 18

2.1.4 MFC's Use of Inheritance 21

2.1.5 Wizards: Guiding the Programmer through the Framework 23

2.1.6 A Summary of MFC 24

2.2 Visual Basic — A Domain Adapted Language for Programming GUIs 24

2.2.1 An Example 25

2.2.2 Delegating Work with VB 26

2.2.3 Pros and Cons of VB 26

2.3 MFC vs VB Conclusion 28

2.4 An Experiment 29

2.4.1 The Design 30
2.4.2 Translation 32
2.4.3 Future Improvements to the Widget Language 32

3 An Attribute Grammar Tool **34**
3.1 Introduction to Attribute Grammars 34
3.1.1 An Example 35
3.1.2 An Overview of the AG 36
3.1.3 The Attribute Definition Functions 39
3.2 Current Features of the AG Tool 41
3.2.1 Positive Features of the AG Tool 42
3.2.2 Negative Features of the AG Tool 42
3.3 Previous Work on Making AGs Modular 43
3.3.1 Pattern AGs 44
3.3.2 Remote Attribute Access 45
3.3.3 Symbol Computations 46
3.3.4 Inheritance 46
3.4 Modular AGs: A Case Study 47
3.4.1 The Base Language 48
3.4.2 The Language Extension 49
3.4.3 Modularising the Translator 51
3.4.4 Modularisation Difficulties 52
3.5 Further Work: A Module System for Attribute Grammars . . 53
3.5.1 Module Interfaces 54
3.5.2 Improved Abstraction Facilities 55

4 Conclusion **56**
4.1 Research Goals 56
4.1.1 Secondary Goals 57

A Adding Exceptions to an Imperative Language **59**
A.1 The Base Language 60
A.1.1 The Grammar 60
A.1.2 `Imperativelib.mli` 61
A.1.3 `Imperativelib.ml` 61
A.1.4 `Internalname.mli` 63
A.1.5 `Internalname.ml` 63
A.1.6 `compile.ml` 64

A.1.7 Attribution Rules 64
A.2 The Extension 69
A.2.1 The Grammar Extensions 69
A.2.2 Attribution Rules 70

Chapter 1

Introduction

Since the invention of the computer, a surprisingly large number of programming languages have been created. Unsatisfied with the mainstream languages that represent the most popular programming paradigms, people continue to create variations on a theme. The reason for this is that the mainstream languages are often found to be clumsy when used in specialised problem domains. Rather than struggle with an unsuitable mainstream language, programmers often find that a better long term solution is to create a specialised programming language. The principal example of this that will be used in this dissertation is the Graphical User Interface (GUI) domain. As we shall show in Chapter 2 (page 15), Graphical User Interface programming can be a slow and difficult process. Hence a number of specialised languages have appeared. Tcl/Tk [23], Visual Basic [31] and Delphi [1] are just a few examples.

In this dissertation we wish to investigate whether language design *reuse* could aid the construction of specialised languages. Therefore, we ask the following question:

> *Can we create new programming languages by composing reusable language components?*

We hope to answer this question in the context of GUI programming. At present, it seems that the best available method of reuse is "cut 'n paste". We intend to research whether a module system could be designed such that language features could be described as reusable modules. These modules could then be easily composed to add new features to a language.

In this chapter we discuss an area of language design that could benefit greatly from language design reuse. This is the area of Domain Adapted Languages: a category which we shall explain in Section 1.1 below. We suggest that such languages could be created by adding Domain Specific Language Extensions to a reusable base-language. In this chapter, we also describe a concept invented by Simonyi [26], called "Intentional Programming" (IP). If languages can be described as compositions of reusable language components, then it is conceivable that a programming environment could be created in which the programming language could be reconfigured by the programmer. This would enable the programmer to alter the language to suit the task at hand. Such an environment would be called an Intentional Programming Environment. In this chapter we describe the goals of Intentional Programming. We also describe the subset of those goals that we intend to achieve with our research.

In Chapter 2 (page 15), we discuss GUI programming. We show that it is a good example of an area in which Domain Adapted Languages can be beneficial. We describe a design and implementation for a Domain Specific Language Extension for GUI programming.

In Chapter 3 (page 34) we discuss Attribute Grammars and show that they form a good specification language for reusable language components. An outstanding research issue which we hope to address is adding a module system to Attribute Grammars. We illustrate the need for a module system with an example, given in Appendix A (page 59).

Finally, in Chapter 4 (page 56) we present our research plan.

1.1 A Categorisation of Programming Languages

Domain-specific programming languages are languages that are intended to be used in a particular problem domain. They contrast with general purpose languages (GPL), such as C, Pascal and ML. In this dissertation, we distinguish between two kinds of specialised language: Domain Specific Languages (DSL) and Domain Adapted Languages (DAL). A DSL is a language that is of little or no use outside its chosen domain. A good example is the language used by the Make utility [6]: it is very good at describing dependency relations, but it could not be used to write (say) a calculator application. On the

other hand, a DAL is a language that is intended for a particular domain, but has the capabilities of a GPL. An example is the language LOGO [10]. Although LOGO is adapted to the domain of controlling "turtles", it can also be used as a GPL.

1.2 Pros and Cons of Domain Specific and Domain Adapted Languages

When applied to their intended domain, DSLs and DALs can offer a number of important advantages:

1. Clarity of notation. DSLs can offer appropriate notation for describing problems in the problem domain.

2. Domain specific error-checking and error-messages.

3. Domain specific optimisations.

The YACC parser generator [12] is a good example. YACC is a DSL for programming parsers. Its input is a grammar description in Backus-Naur form. Its output is a program which is specifically designed for parsing that grammar. YACC demonstrates each of the above advantages:

1. YACC provides good notation for describing grammars.

2. YACC checks grammars for ambiguity.

3. YACC compiles parser specifications to code that uses an efficient push-down automaton.

DSLs and DALs can also have disadvantages:

1. The tool support for DSLs and DALs may be inadequate. This is because such languages are often not very widely used.

2. The problem domain may shift. If the DSL/DAL that is being used to solve the problem does not evolve too, then it may become unusable. Therefore a GPL may be more appropriate for a rapidly evolving domain.

3. A DSL/DAL may cause compatibility problems, because it is a non-standard programming language. This can be an issue if the programming problem spans several domains. For example, consider the domain of compiler programming. A compiler consists of more than just a parser. Therefore YACC implementations are always designed to have a good interface to a GPL such as C or ML.

4. Learning to use a new language is a drain on programmer time. It may not be worth introducing a new DAL/DSL unless it is likely to bring significant productivity gains.

The disadvantages listed above are all of an economic nature. They will not arise if enough programmer time is allocated to the maintenance of the DSL/DAL. However, the cost of maintaining a DSL/DAL needs to be weighed against the cost of simply using a GPL.

1.3 Domain Specific Language Extensions

Above, we discussed the fact that specialised languages can be costly to develop and maintain. Particularly in the case of Domain Adapted Languages, language design reuse could help to reduce this cost. DALs contain the full functionality of a general purpose language. Therefore, every time a DAL is implemented, redundant work is necessary to implement standard language features like branches and loops. This work could be avoided if it were possible to define Domain Specific Language *Extensions*. Rather than being an entire new language, a DAL would be created by adding domain specific features to an existing general purpose language. A DAL created in this way would inherit the languages features and tools of the base language, thereby both saving work for the implementor and improving the quality of the result. In addition, it would make the DAL easier to learn for programmers who are already familiar with the base language. Moreover, there wouldn't be a need for a foreign function interface for interfacing between the two languages.

1.4 Intentional Programming

We suggested above that it might be possible to create DALs by adding domain specific language extensions to a GPL. Simonyi [26] has suggested the

idea of a programming environment in which this can be done. This environment, known as an "Intentional Programming" (IP) environment, would simultaneously support both ordinary programming and meta-programming. In this model, ordinary programming is the use of the DALs and the meta-programming is the creation of new language extensions. A meta-program that implements a language extension is known as an "Intention".

Simonyi is currently leading a project at Microsoft to create an implementation of IP. The project has successfully created an advanced development environment in which the programming language is not strictly fixed and can be extended. Unfortunately, meta-programming currently has to be done in a language based largely on C. This often makes meta-programming a slow and error-prone process. Most importantly though, due to the meta-programming language being fixed, it is not possible to experiment with new formalisms for describing and composing language extensions. Therefore, in this dissertation we propose that a lightweight tool should be used for experimenting with language extensions. We describe this in more detail in Chapter 3 (page 34).

1.5 Requirements of an Intentional Programming Environment

Intentional Programming is still a developing concept. Therefore it needs to be made clear what the goals of IP are. It also needs to be clear which of these goals will be disregarded by the lightweight tool proposed in Chapter 3 (page 34). The requirements of an Intentional Programming Environment are:

1. The environment provides a meta-programming language for describing Intentions.

2. The environment can compose intentions to create new DALs. A translator for the new DAL should be automatically generated.

3. The environment should not allow Intentions to be composed unless they are 'consistent'. We define consistency as follows: A proposed combination of Intentions is inconsistent if the environment cannot automatically generate a deterministic translator for the resulting lan-

guage. We might add to this definition the requirement that the translator should also be terminating.

4. The environment can be used for both meta-programming and ordinary programming. The system should be bootstrapped in the sense that the meta-programming language can also be extended with new intentions.

5. The environment is not restricted to any particular language. The base language that is being extended can be any language.

The IP system under development at Microsoft satisfies some, but not all of these requirements. Most importantly, a consistency check is not performed. Incompatibilities between intentions are sometimes discovered at compile time, but this is not guaranteed. The compilation process is not guaranteed to terminate. In its current form, the system also doesn't satisfy the fourth point. This is because the meta-language is fixed. This situation will change in future versions of the system.

The aim of the tool described in Chapter 3 (page 34) is to satisfy all the points except the fourth. Unlike Microsoft IP, we are placing a very low emphasis on creating an integrated environment. Microsoft IP is a sophisticated environment in which editing is done with a structure editor. This means that there is no parsing phase, so grammar ambiguity is not an issue. This can make it easier to compose intentions, but it is a heavyweight solution. We intend to use conventional parsers.

1.5.1 Ecology

Simonyi [26] has a vision that in an IP environment as described above, an "ecology" will emerge. This ecology will be an ecology of Intentions. In an IP environment, programmers can decide which Intentions they want to program with. In other words, Intentions will be competing for survival. Simonyi hopes that this will lead to a gradual evolution of programming languages.

We believe that the consistency check described above is crucial in an ecological environment. Without it, a programmer could write a large program before realising that two incompatible Intentions have been used.

The promise of an ecology sounds similar to the promise of reusability in object oriented programming. The promise was that "reusable" objects

could be simply thrown together to create new applications. However, experience has shown that "reusable" objects do not always compose as easily as one might hope. This is shown by Schneider and Nierstrasz [25]. They explain that in practice, either "glue-code" needs to be written to enable the composition or the objects all need to be members of a single carefully designed toolkit. We predict that the same problem will arise in Intentional Programming. When adding an Intention to a language, glue-meta-code will need to be added to describe the interface between the Intention and the language. Therefore we think an Intention will not be successful in the ecology unless it satisfies the following criterion:

> *The Intention can be added to a number of different languages with a minimum of glue code.*

When we design Intentions, the above criterion will be one of our main design goals.

1.6 Previous Work

Three areas of previous work are of particular relevance to Intentional Programming. Firstly, work has been done to make programming languages extensible by adding macros to the language. Secondly, work has been done to make compilers extensible by adding hooks which allow the programmer to customise the compilation process. Thirdly, work on attribute grammars is of interest, because we intend to use attribute grammars as our implementation technique. Previous work on attribute grammars will be discussed in Section 3.3 (page 43).

1.6.1 Macros

Kernighan and Ritchie describe how the C preprocessor [16, pages 89–91] can be used to add language extensions to C. They use it to define `forever`, which is an infinite loop. They also use it to create a polymorphic implementation of `max`, despite the fact that C does not support polymorphism. The C preprocessor is a purely text-based tool. It simply replaces every occurrence of a macro with its expansion. This means that macros can be defined which generate syntactically or semantically incorrect code. They can also lead to subtle errors which are not spotted by the C compiler. Kernighan and

Ritchie give an example of this: `max(i++, j++)`. If `max` were a conventional function, `i` and `j` would be incremented once after `max` had been evaluated. However, due to the implementation of the macro, the larger of `i` and `j` will be incremented twice.

Some of the errors that occur with text-based macro processors can be avoided by macro processors that interact with the compiler. Syntax macros use this technique to prevent the programmer from defining a macro which will generate syntactically incorrect code. Syntax macros were developed independently and concurrently by Leavenworth [20] and Cheatham [2]. They preserve syntactic correctness by specifying their interaction with the grammar of the language. This is done by specifying the non-terminal type of the macro and its parameters. For example, we could define a "while" macro with the following signature:

statement **macro** while(*expression*) do *statement*

This means that the macro expects two parameters. The first should be an expression of the language and the second should be a statement. The macro is only valid in a statement context. Depending on their design, syntax macros can cause complicated parsing problems. If macros are allowed to be embedded in the source text, then the parser must be able to dynamically reconfigure when it encounters a new macro definition.

1.6.2 Extensible Compilers

Engler [4] describes a system called Magik, which allows programmers to modify the compilation process. Magik is a customised version of the lcc C compiler [7]. It allows the programmer to write routines that modify the internal abstract syntax tree during the compilation process. There is no way of adding new syntax to the language, but language extensions can be created by treating certain function calls and data-structures as special cases. For example, an extension could be written that partially evaluates calls to `printf`. Magik applies the user extensions at every function call and data structure definition that it encounters. The extensions have access to any data flow information computed by the compiler. The extensions are allowed to modify the internal abstract syntax tree. Magik loops through the extensions until no more modifications occur. Engler does not make any guarantees about termination of the compilation process. He also does

not discuss the issues involved in composing extensions. These issues are important in Intentional Programming, because it is important that separate Intentions do not interfere with each other.

Stichnoth and Gross [27] use a technique called "code composition" to implement compilers. Compilers often translate each operation of the input language into a fixed sequence of operations in the target language. Optimisations are subsequently applied to generated code. When the input language becomes more complicated, the sequence of target language operations corresponding to each operation of the input language can become much more complicated. This can cause problems if the code sequences are hard-coded in the compiler. Therefore compiler-writers often shift to the runtime library approach. In this approach, the input language operations are translated to calls to library functions. This allows the code sequences to be defined externally to the compiler. However, it makes it impossible to apply optimisations to the generated code. Code composition allows the code sequences to be defined externally to the compiler. During compilation the code sequences are imported and composed to generate code. Using this technique, optimisations can still be applied to the generated code. The authors' system, Catacomb, provides a simple programming language for describing *code templates*. These templates generate the chunks of code that will be composed. Catacomb also applies optimisations to the code. The authors note that a two-phase approach, whereby first the templates are expanded and then the optimisations are applied, is the most straightforward way of achieving this. Catacomb uses a one-phase approach. This allows the templates to benefit from the optimisations that can be applied before they are expanded. Unfortunately the authors do not explain how their implementation works.

Chapter 2

GUI Language Extensions

In this chapter we describe the issues involved in programming Graphical User Interfaces (GUIs). GUI programming is shown to be a domain that is well suited to the use of domain adapted languages. The design of a language extension for GUI programming is proposed as a case study in the use of Domain Specific Language Extensions.

GUI programming has become an important field of modern industrial programming. Many companies use GPLs to program GUIs, but DALs are also widely used. This chapter gives a detailed comparison of two GUI development tools. The first is the Microsoft Foundation Class (MFC) library. MFC is an object-oriented toolkit that is implemented in C++. It serves as an example of the GPL approach. The second tool is Visual Basic, which is a domain adapted version of the language Basic. Visual Basic contains specialised language features for programming GUIs. Both tools are products of Microsoft. They are designed to produce GUI applications that run on the Microsoft Windows operating systems.

2.1 The Microsoft Foundation Classes

The Microsoft Foundation Classes (MFC) [19] are currently one of the most widely used toolkits for GUI development on Microsoft Windows operating systems. MFC is implemented as an object oriented framework in C++. Therefore it serves as an example of the use of a GPL to program GUIs.

MFC's popularity stems from the following advantages:

Look and Feel. Applications written in MFC have the Microsoft look and

feel. This means that the application will automatically have a familiar layout and behave in a way that Windows users expect.

Large Toolkit. MFC is a large library, so the programmer has access to a wide selection of components.

Uses C++. C++ is currently the industry standard programming language. As such, it is very well supported with good development tools. MFC ships with Microsoft's C++ compiler: Visual C++.

However, MFC is hard to learn and difficult to use. This is because of the complicated structure of MFC applications. This structure is discussed below.

2.1.1 The MFC Application Framework

MFC is not a simple library of C++ classes. It is an application *framework*. This means that the MFC classes will only work correctly when used together in the intended way. When used correctly, the framework coordinates the communication between the application and the operating system. It provides a bridge between the lower-level callback-oriented Win32 API and MFC's higher-level object-oriented structure. For an explanation of the Win32 API, see Petzold [24].

The key components of the MFC framework are the Document-View structure and the Message Mapping system. We describe them below. We also explain MFC's use of inheritance, which is sometimes a little confusing. Finally, we describe "Wizards". These are tools that help the programmer to use the framework.

Frameworks

Frameworks are an object-oriented technique that aid software reuse. As described by Johnson [11], a framework is an abstract design for a particular kind of application. This design consists of a number of abstract classes, which the application should implement. Frameworks are often used in conjunction with Design Patterns. A Design Pattern is a purely abstract description of an object-oriented design structure. It describes how a particular structure of classes can be used to achieve a design goal. An example is the Model-View-Controller pattern [18] (described in Section 2.1.2 below).

Gamma *et al* [8] describe many more. The distinction between Frameworks and Design Patterns is that Frameworks tend to be more concrete than Design Patterns. Whereas a Framework specification often includes some code and class interfaces, a Design Pattern specification tends to be given as an informal prose description. By their nature, frameworks and design patterns are not part of the programming language. They exist merely as a set of guidelines that are not enforced by the type-checker. As such, they may make good candidates for encoding as intentions.

2.1.2 The Document-View Architecture

The top-level structure for an MFC-application is based on the Model-View-Controller (MVC) design pattern [18]. MVC is a technique for designing user interfaces. It consists of three kinds of objects. The Model represents the data. The View represents the screen presentation of the data. The Controller defines the way that the View responds to user input. The idea is that several Views of one Model can be simultaneously active. The design pattern ensures that the Views are all kept up to date with the Model. MFC's "Document-View" architecture provides the functionality for keeping the Views up to date with the Model. It also provides the user interface for creating and destroying Views and Models. For example, it automatically adds menu options for opening documents and viewing them in a window. The architecture is known as the Document-View Architecture, because the concept of a Controller is absent. The Controller functionality is contained in the View.

To create the Document-View structure, the programmer has to create the following classes:

`CMyApp` (inherits `CWinApp`) This class represents the main application. Exactly one global instance should be created.

`CMainFrame` (inherits `CFrameWnd`) This is the main window of the application. It contains the menu, toolbars and status bar. Views of documents appear as sub-windows inside it.

`CMyDoc` (inherits `CDocument`) This is the document component of the design pattern. Therefore it should only contain information about the contents of the document. It is not concerned with the layout on the screen.

The programmer can define different classes for every document-type that the application supports.

`CMyView` (inherits `CView`) This is the sub-window that displays the document. It provides the user interaction for editing the document. Multiple views of the document can appear on the screen. The Document-View architecture keeps them consistent with the document (and therefore with each other). More than one view type can be defined for one document type.

An MFC application does not have a `main()` function. Instead, the entry point of the program is a single global instance of `CMyApp`. `CMyApp` should have a method `InitInstance()` which does program initialisation. `InitInstance()` also makes the connection between the four components described above. This is done by the following segment of code, which must appear in `InitInstance()`:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CMyDoc),
    RUNTIME_CLASS(CMainFrame),
    RUNTIME_CLASS(CMyView));
AddDocTemplate(pDocTemplate);
```

The above code is the 'magic' step that ties together the four components. MFC programmers need to know that this call should be made and they need to know exactly where to make it. They also need to know what the side effects of this call are. For example, it sets the value of `m_pMainWnd`, which is a data member of `CMyApp`. From now on, `m_pMainWnd` should be used to access `CMainFrame`.

2.1.3 The Message Mapping System

The Object-Oriented programming paradigm is well suited to programming GUIs. This is because graphical widgets are very object-like in nature. Unfortunately there are a few technical issues involved in encapsulating the Win32 API with C++ classes. In this Section, we explain the problem and describe MFC's solution: message maps.

Win32 Callback Functions

Petzold [24, page 62] explains that in the Win32 API, each widget has a callback function, which determines its behaviour. The callback function is called by the API whenever there is a message waiting for the widget. Typical examples of those message are mouse-clicks, re-paint notifications and re-size messages. The signature of the callback function is as follows:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
```

The parameter `hwnd` identifies the widget. This parameter is used to distinguish widgets when many instances of one widget type have been created. `HWND` is an abstract type, known as a *handle*. The parameter `message` contains the message ID. Some messages need to be accompanied by additional information. For example, the re-size message needs to specify the new size of the widget. The parameters `wParam` and `lParam` are used for this purpose.

At the top level, the callback function usually consists of a `switch` statement. This statement does a case analysis on the `message` parameter and picks out the messages which are of interest to the widget. If the message is not recognised by the widget then a Win32 function called `DefWindowProc()` is called. `DefWindowProc()` provides the default behaviour for these messages.

This use of callback functions to communicate messages has the following important benefits:

1. It is easy for widgets to ignore messages that are irrelevant to them. This ensures that the code for uncomplicated widgets is simple and lightweight.

2. The Windows OS can introduce new messages without old applications needing to be recompiled. This is because the callback functions simply ignore messages which they don't recognise.

3. The Windows OS can change the default behaviour of widgets without old applications needing to be re-compiled. It does this by changing the implementation of `DefWindowProc()`.

4. User-defined messages can be posted to widgets via the OS. A range of message IDs is reserved for exactly this purpose.

Below, we explain why some of these advantages are lost if widgets are modeled as C++ objects.

Modeling Widgets as Objects

The Object-Oriented style of describing widgets is as follows. Every widget is an instance of a class. Suppose the object is called `MyWidget`. `MyWidget` should inherit an object called (say) `BaseWindow`. The `BaseWindow` object includes methods for every message that the OS could send to the widget. These methods represent the default behaviour of a widget. `MyWidget` redefines the methods corresponding to the messages that it is interested in. The other methods inherit the default implementation from `BaseWindow`.

Inheritance in C++

Unfortunately, the inheritance mechanism in C++ is not well-suited to the requirements of widget programming. In C++, inheritance is implemented with v-tables. Therefore, both `BaseWindow` and `MyWidget` would need to have v-tables that contain entries for every message supported by the Windows OS. This implementation would negate the advantages that the Win32 callback mechanism offers, because it is not flexible when the list of messages changes. In particular it is not clear how it would deal with user-defined messages. Kruglinski *et al* [19, pages 26,27] point out that the v-tables would also be inefficient in terms of memory usage, because they would each contain over 100 entries. It is debatable whether this consideration is of any relevance on modern hardware.

Message Maps

MFC uses C++ classes to represent widgets, but does not use the inheritance mechanism to implement messages. Instead a callback function is used to receive messages from the OS and invoke the appropriate methods in the Widget class. The code for the callback function is generated by macros. Suppose for example that the class `MyWidget` needs a message handler for mouse-clicks. Kruglinski *et al* [19, page 48] describe the code that is needed. Firstly, `MyWidget` requires an appropriate member function. Its signature is:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point)
```

Secondly, a "message map" needs to be declared for `MyWidget`:

```
BEGIN_MESSAGE_MAP(MyWidget, CView)
    ON_WM_LBUTTONDOWN()
    // other message map entries
END_MESSAGE_MAP()
```

This message map should contain entries for every message that the widget responds to. (Only the button-click message has been shown in the above code.) Some entries (like `ON_WM_LBUTTONDOWN` above) have specialised macros. Other entries need to use the more general `ON_MESSAGE` macro.

The message map macros above expand to the code for the callback function. Another macro is needed to create the signature. It should appear in the class header for `MyWidget` as follows:

```
DECLARE_MESSAGE_MAP()
```

2.1.4 MFC's Use of Inheritance

MFC's message maps are designed to bypass the C++ implementation of inheritance. However, MFC also uses conventional C++ inheritance. This means that two orthogonal forms of inheritance are being used simultaneously, which can lead to some confusion. In this Section, we describe the most important ways in which MFC uses C++ inheritance.

The `CWnd` base-class

Any class that has a message map should inherit `CWnd` or one of its derivatives (such as `CView` or `CFrameWnd`). This is because `CWnd` contains code and data members that are reused by all widgets. Examples of this are the code for registering the widget with the OS and the code for implementing the message map.

Messages that *do* use C++ inheritance

Occasionally, there are exceptions to the rule that C++ inheritance is not used for message maps. A prime example is found in the class `CView`. `CView` is a class that is used in the Document-View architecture. `CView` inherits `CWnd` and adds methods for interfacing with the architecture. Any view

window in the architecture should inherit CView. CView has a method called OnDraw() which can be overridden. This method is called when the Widget receives a paint message from the OS. Kruglinski *et al* [19, page 40] explain why the paint message is treated differently to other messages: it allows the framework to implement printing behaviour. To print the document, the framework calls OnDraw() with the paint context set to the printer rather than the screen.

Window Sub-classing

C++ inheritance is also used to achieve an effect known as Window sub-classing. Window sub-classing is the reuse of another widget's behaviour with minor modifications. The modifications are made by intercepting some of the messages that the widget receives. Petzold [24, page 393] explains that this can be done in the Win32 API by writing a callback function which handles one or two messages and forwards the rest to the callback function of a different widget type. For example, this technique could be used to create a numeric edit-box. That is, an edit-box that will not accept non-numeric characters. Window sub-classing is used to intercept and delete messages that correspond to non-numeric keystrokes.

Kruglinski *et al* [19, page 376] explain how window sub-classing is done in MFC. Suppose a numeric edit-box is required. The class for the standard edit-box is CEdit. Therefore a new class is created which inherits CEdit. Suppose the new class is called CNumericEdit. CNumericEdit should be given exactly one message map entry for the WM_CHAR message (which is sent when a key is pressed in the edit-box). All other messages are handled by CEdit. The handler function in CNumericEdit filters the stream of characters, passing only the digits on to CEdit:

```
void CNumericEdit::OnChar(UINT nChar, UINT nRepCnt,
                          UINT nFlags)
{
    if(isdigit(nChar))
       CEdit::OnChar(nChar, nRepCnt, nFlags);
}
```

Window Subclassing can be a dangerous technique, because the interface between the OS and system components such as the edit-box is not clearly

defined. It could change without notice in future releases of the OS. For example, the above code for `CNumericEdit` is no longer complete since the introduction of the clipboard. A user of the widget could still enter a non-numeric character by cutting and pasting it.

2.1.5 Wizards: Guiding the Programmer through the Framework

As described above, MFC has a number of conventions that are not enforced by the C++ compiler. In particular, the programmer could easily make a mistake when creating the Document-View structure. Microsoft's Visual C++ programming environment tries to keep the programmer on the rails with facilities called Wizards. A Wizard is an editing tool. The most important Wizard is the Application Wizard. It creates an operational initial application that has the Document-View structure already in place. The second most important Wizard is called the Class Wizard. It is useful for adding message handlers to classes. The problem with adding message handlers is that it involves editing the code in three different places: the class header, the method definition and the message map entry. The Class Wizard automatically finds these three locations and updates them.

The Wizards have two important disadvantages:

- *The Application Wizard is non-reversible.* If a wrong option was chosen during the execution of the Wizard, then this cannot be corrected in hindsight. The only solution is to generate a new application and start again. This can be very inconvenient, because the code generated by the Wizard and the code written by the programmer is often not cleanly separated into different files.

- *The code needs to contain markers, so that the Class Wizard knows where to edit the code.* MFC code frequently contains comments like:

  ```
  // {AFX_MSG_MAP(MyWidget)
  ```

  These comments are necessary for the Class Wizard to edit the code. Therefore the programmer must be careful not to edit these areas of the code.

In summary, Wizards help the programmer to write MFC applications, but they are really a symptom of the fact that MFC applications are unnecessarily complicated to write.

2.1.6 A Summary of MFC

As we said at the start of the chapter, MFC is an example of a GUI toolkit which is implemented in a GPL. Although MFC is a large and versatile toolkit, it can also be difficult to use. To use MFC, the programmer needs to become familiar with its application framework. This framework is a set of guidelines. These guidelines are not enforced by the C++ compiler, so the programmer needs to take great care. The programmer also needs to be familiar with some of the internal implementation details of MFC. Otherwise the Document-View architecture and the Message Mapping system make very little sense.

Facilities called Wizards are provided to make MFC programming easier. They are editing tools which take some of the work out of entering MFC programs. They help to enforce the application framework by generating skeleton code, but they are no substitute for an automated checker.

2.2 Visual Basic — A Domain Adapted Language for Programming GUIs

Visual Basic (VB), an adapted version of Basic, is rapidly gaining popularity as a prototyping and scripting language. It is also being used to develop production code.

VB is a customised language that has built-in support for GUI programming. Programmers find it significantly easier to learn than MFC. Code written in VB is concise and isn't confusing like MFC. The most important difference between VB and MFC is that in VB, widgets have a clear and simple interface specification. It consists of the following three kinds of elements:

events: An event is a message originating from the widget. The owner of the widget can listen for events by providing a handler function.

Example: An edit-box widget has a "text-changed" event.

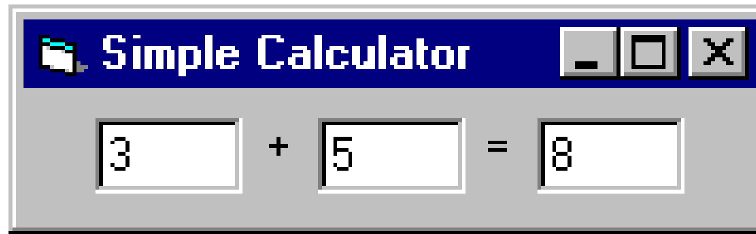

Figure 2.1: A Screen Shot of the Calculator

properties: A property is a variable associated with a widget. Properties can be accessed by the widget's owner. They can be read-only or read-write.

Example: An edit-box has a "current-text" property.

methods: These are methods which can be called by the widget's owner.

Example: An edit-box has a "set focus" method, which makes it the active window for user input.

2.2.1 An Example

In this section, we illustrate VB's model of GUI programming with an example. The example is a simple calculator program. A screen shot of the program is shown in Figure 2.1 (page 25). The user interface contains three text-boxes which we shall refer to as T1, T2 and T3 (numbered from left to right). When the user edits one of the text-boxes, the program alters another such that the invariant T1 + T2 = T3 is maintained.

The code for the program is given in Figure 2.2 (page 27). Here we see the handler functions for the widget events. For example, `Form_Load` is called when the main window (called "Form") is created. The code also contains a conventional datatype definition (`Maybe`) and a number of conventional subroutines. Absent though, are the definitions that were made "visually". Part of the Visual Basic development process is to design the user interface with a graphical tool. For the user interface of this program, we drew a Form and added three text boxes and two labels (containing the symbols + and =).

When the user edits one of the text-boxes, the corresponding `Text_Change`

handler is called. The handler calls `readnum` to interpret the contents of the box. The user may not have entered a valid number, so we use the `Maybe` datatype to represent the contents of the box. We have written special routines `plus` and `minus` for performing arithmetic on values of type `Maybe`. These are used to reestablish the invariant and then `writenum` is called to update one of the text-boxes.

It should be clear from this example that GUIs can be described very concisely with VB. Widgets are easy to manipulate, because their properties, methods and events are easily accessible. Widgets can also be passed as parameters, which is why we can define abstractions like `readnum` and `writenum`.

2.2.2 Delegating Work with VB

Although VB is well-suited to programming GUIs, one may prefer to program the core of a program in a different language. Visual Basic makes this possible by offering a good foreign function interface. The foreign function interface works with Microsoft's Component Object Model (COM). COM is a protocol for building applications from dynamically linked components. In fact it is specifically designed for connecting components that were written in different programming languages.

2.2.3 Pros and Cons of VB

Pros

VB provides built-in language support for GUI programming. As demonstrated in Section 2.2.1 (page 25), this makes GUI programming easy and concise.

VB provides a number of useful tools for GUI programming. The Object Browser allows the programmer to search the events, properties and methods of every available widgets. A visual editor is provided for building GUIs graphically. Context sensitive help is provided to explain the use of the libraries.

VB is based on Basic. As the name suggests, Basic is an uncomplicated programming language. This helps to make Visual Basic easy to learn.

Visual Basic has a good foreign function interface. This means that the core of the application can be written in a different programming language.

```
Private Type Maybe
  value As Long
  valid As Boolean
End Type

Dim t1 As Maybe
Dim t2 As Maybe
Dim t3 As Maybe

Private Sub Form_Load()
  t1.valid = False
  t2.valid = False
  t3.valid = False
  Text1.Text = ""
  Text2.Text = ""
  Text3.Text = ""
End Sub

Private Sub Form_Resize()
  Dim w As Integer
  Dim h As Integer
  w = Form1.ScaleWidth / 10
  h = Form1.ScaleHeight / 5
  Text1.Move w, h, 2*w, 3*h
  Label1.Move 3*w, h, w, 3*h
  Text2.Move 4*w, h, 2*w, 3*h
  Label2.Move 6*w, h, w, 3*h
  Text3.Move 7*w, h, 2*w, 3*h
End Sub

Private Sub Text1_Change()
  readnum Text1, t1
  plus t1, t2, t3
  writenum Text3, t3
End Sub

Private Sub Text2_Change()
  readnum Text2, t2
  plus t1, t2, t3
  writenum Text3, t3
End Sub
```

```
Private Sub Text3_Change()
  readnum Text3, t3
  minus t3, t2, t1
  writenum Text1, t1
End Sub

Private Sub readnum(T As textbox, _
                    x As Maybe)
  On Error GoTo Seterror
    x.value = CLng(T.Text)
    x.valid = True
  Exit Sub
  Seterror:
    x.valid = False
End Sub

Private Sub writenum(T As textbox, _
                     x As Maybe)
  If x.valid Then
    T.Text = CStr(x.value)
  Else
    T.Text = ""
  End If
End Sub

Private Sub minus(a As Maybe, _
                  b As Maybe, _
                  result As Maybe)
  result.value = a.value - b.value
  result.valid = a.valid And b.valid
End Sub

Private Sub plus(a As Maybe, _
                 b As Maybe, _
                 result As Maybe)
  result.value = a.value + b.value
  result.valid = a.valid And b.valid
End Sub
```

Figure 2.2: The Visual Basic Code for the Calculator Application

Cons

Although VB's visual editor is often very convenient, it can also be a hindrance. The problem is that the use of the visual editor is not optional. This is because certain operations provided by the visual editor do not have a textual counterpart. For example, the controls on a Form must be added visually. There is no language construction for adding a control to a form. An implication of this is that it is impossible to add a new control at run time. In fact, as described in the Visual Basic 6.0 Programmers Guide [31, page 155], the `New` keyword for dynamic object creation is not applicable to controls. Visual editing also has the disadvantage that searching the source code is harder. This is because some of the definitions are now recorded in a visual format which cannot be mechanically searched.

Being a language which is easy to learn, VB is not necessarily as powerful as some other programming languages. For example, faster programs can be written in C, because C is a lower level language. ML has a more rigorous type-system than VB, because it is static, rather than dynamic. So VB is often used in conjunction with another programming language. The COM interface makes this possible. However, this adds work to the programming task, because COM programming can be complicated.

2.3 MFC vs VB Conclusion

We have described two GUI programming environments. MFC and VB are both products of Microsoft, aimed at Windows GUI development. However, MFC is implemented as an object-oriented framework in C++, whereas VB is a DAL.

MFC is a powerful library which is based in the industry standard language C++. To use it the programmer must be familiar with its application framework though. In particular the programmer needs to know about the Document-View architecture and Message Maps. The 'rules' of MFC's framework are actually only guidelines: they are not enforced by the C++ compiler. Therefore MFC programming must be done with care.

VB has much better facilities for GUI programming than MFC. This is because the language has built in support for widgets. Most importantly, widgets have interfaces which are type-checked by VB. Some programming in VB is done "visually". This can be useful for rapidly designing a user-

interface. Unfortunately, some operations which can be performed with the visual editor do not have textual counterparts. This can make the language inflexible, because those operations are not abstractable. VB uses COM to provide a foreign function interface. This allows the core of the program to be written in a different language. Programming a COM interface is non-trivial though, so this adds complexity to the program.

2.4 An Experiment

To test whether Domain Specific Language Extensions are a feasible idea, they need to be applied to a sizeable example. GUI programming lends itself as a suitable area, because it has been shown to gain from domain specific notation. It is also an important field in modern computing. The aim of the experiment is not to improve upon current GUI programming systems. Instead, the aim is to provide similar facilities via the mechanism of language extensions. VB's model of GUI programming seems successful, so we will use a similar system in the language extension.

As described for VB, a widget has an interface consisting of events, properties and methods. Internally, a Widget can be constructed by composing sub-widgets. The events, properties and methods of the widget can then be defined in terms of the events, properties and methods of the sub-widgets.

Initially the widget language will be simplistic in a number of ways. A number of possible improvements to the language are listed in Section 2.4.3 (page 32). In the initial design, the following simplifications will be made:

1. The sub-widgets of a widget must be statically and individually declared. In other words, no dynamic creation of sub-widgets and no arrays or lists of sub-widgets. This removes the need to interact with type-constructors of the base language (such as array).

2. Window sub-classing is not possible. As described in Section 2.1.4 (page 22), window sub-classing can be a dangerous technique. It could only be done properly if the widget interface also specified the interface between the widget and the OS.

3. The language only describes widget construction. Other Intentional forms such as the Document-View architecture are not yet covered.

Below, we describe the initial design for the language extension.

2.4.1 The Design

The language extension consists of three parts. Firstly, notation is needed for describing new widgets. This includes notation for defining the behaviour of a widget and notation for describing its interface. Secondly, new statements and expressions need to be added for accessing the properties and methods of widgets. Thirdly, a layout language is needed for describing the graphical layout of the program. These elements are described below.

Widget Definitions

A Widget Definition introduces a new kind of widget. Instances of this widget may then be used by the application. The definition consists of the following elements:

name An identifier for the new widget type.

initialise This is a special method that is executed when the widget is created. Its parameters contain information that must be supplied by the owner when the widget is created.

events This section lists the events that the widget can fire. The conditions that lead to the firing of an event are not listed here. Instead events are triggered as side-effects of methods or when a property changes value. Events can be parameterised. This allows the event to carry extra information.

properties This section lists the properties of the widget. Properties are publicly accessible data-members of the widget. An event can be associated with a property, so that the event will automatically fire if the property changes. The value of the property is updated by side-effects elsewhere in the widget's code. If the property is specified to be writable, then its value can also be updated by the widget's owner.

methods Methods are defined by functions or procedures written in the base language. They can have side-effects that update the values of properties and trigger events.

subwidgets This is the list of subwidgets that the main widget contains. Recall that they must be listed individually and statically. The main

widget and the subwidget can communicate via the events, properties and methods of the subwidget.

handlers A handler is a procedure or function which is associated with an event of one of the subwidgets. When that event fires, the handler is called.

utilities Utilities are functions and variables written in the base language. They are not accessible from outside the widget.

Widget Interfaces

The interface to a widget consists of the signatures for its events, properties and methods. We shall refer to this as the implementation independent interface to the widget. Internally the translator may need to record more detailed information about the widget. For example, if we are translating to MFC, the implementation needs to know which message number is used by each method and event. So there need to be two versions of the interface: one dependent on the implementation and one not. The implementation specific interface is generated automatically during translation. The programmer should not be concerned with implementation specific details.

Statement-Level Language Features

The code for the methods and handlers is written in the base language. This code needs to be able to access the values of properties and call methods of the sub-widget, so new statements need to be added to the base-language. Property values are a new kind of expression. Method calls are a new kind of statement. Triggering an event is also a new kind of statement.

The Layout Language

One of the restrictions that we have placed on the language design is that widgets cannot be dynamically created: we need to provide a static means by which the application can create its user interface. We shall do this by replacing the entry point of the language. The entry point of an imperative language is the position where program execution starts. For example, in a C program it is the `main()` function. In our language, the program is entered via a layout language which specifies the graphical layout of the program.

This structure is analogous to Visual Basic, where the top-level application design is done with the visual editor. Our layout language will have similar features to the visual editor in VB. However, it shall be text based, rather than visual.

2.4.2 Translation

The widget language is defined as an extension of a base language. Therefore, programs written in the widget language are translated by reducing them to programs written in the base language. Widget definitions translate naturally to either modules or classes in the base language. This is because they are separable components that introduce a new datatype and some associated code. The translator needs to apply the following steps to translate the widget:

- Locate and read the implementation specific interfaces for all the subwidgets.
- Translate method calls, event handlers and property accesses that refer to the subwidgets. This is done by replacing those occurrences with implementation specific code written in the base language. This is where the information from the widget interface files is used.
- Generate a module, written in the base language, that contains the code for the widget. Depending on the rules of the base language, an interface for this module may also need to be generated.
- Generate the implementation specific version of the widget interface.
- Translate the layout language to the top-level code for the application. This is the code that creates the user-interface when the application is started.

2.4.3 Future Improvements to the Widget Language

As described in the introduction of Section 2.4 (page 29), a number of restrictions have been made to the widget language. This is to simplify the initial experiment. It is hoped that it will be possible to lift some of these restrictions. Some of the improvements that could be made to the language are the following. They are listed in order of importance:

1. Widgets should become typed objects in the language. They can be created and destroyed in the same way that other variables are. This will remove the need for the sub-widgets of a widget to be statically defined.

2. The language should include support for certain design patterns. In particular, the Document-View architecture should be added.

3. The type-system of the language could be extended, so that window sub-classing can be done safely.

Chapter 3

An Attribute Grammar Tool

The Intentional Programming tool being developed at Microsoft uses a translation technique called reduction. O. de Moor, who leads the Oxford IP project, has noticed the similarity between reduction and Attribute Grammars. Attribute Grammars (AGs) were discovered by Knuth [17]. They are a formalism for associating semantics with syntax trees. AGs are particularly useful for describing compiler front-ends, which is why they are of interest in IP.

In this chapter, we give a brief explanation of Attribute Grammars. An AG tool that we have written is described. It is a lightweight tool that translates AG specifications to OCaml [21] programs. The tool builds on some of the research done by Van Wyk [30]. In this chapter we describe the features of the tool, which are at present fairly basic. Later, we propose some improvements that should make the tool usable as an IP prototype.

3.1 Introduction to Attribute Grammars

An Attribute Grammar is a tool that is used to add semantics to a syntax tree. It does this by annotating the nodes of the tree with attribute values. For example, each node of the tree that represents an expression might be annotated with its type. Each node that represents a new scope might be annotated with the list of new variables that it introduces. These annotations are used in the next phases of the translation process. The attribute values are calculated by applying semantic functions, which are specified by the AG. In this Section, we give a short example to demonstrate how this works.

```
{
   Use x
   Use y
   {
      Dec y
      Use y
      Use x
   }
   Dec x
   Use x
   Dec y
}
```

Figure 3.1: An example input to the AG

3.1.1 An Example

The example given in this Section is taken from “Aspect Oriented Compilers”, by de Moor *et al* [22]. It is an AG for a simple programming language. The language is too simple for meaningful programs to be written in it, but it demonstrates scoping and the use of environments. Figure 3.1 (page 35) shows an example input to the AG. There are essentially three elements to the language: scopes, variable declarations and variable uses. The scope rules of the language are:

- Declared variables are visible anywhere within the scope that they are declared in. (Including before the definition, so `Use x` followed by `Dec x` is allowed.)
- A variable declaration can use the same name as a variable that appears in an enclosing scope. The variable in the enclosing scope becomes invisible within this scope, because the new variable declaration takes priority.
- Uses of variables refer to the declaration that appears in the current scope.

An AG for this language is given in Figure 3.2 (page 36) and Figure 3.3 (page 37). In fact, the AG specification given by those Figures is a valid input to

```
start program : code

token LPAREN, RPAREN, USE, DEC
token <string> STRING

inherit copyable <int> level
inherit copyable <Testtypes.envir> env
synthesise <string> code
synthesise <Testtypes.locstype> locs

program   -> block                  { Main }
block     -> LPAREN stmtlist RPAREN { Block }
stmtlist  -> stmt stmtlist          { Stmtlist2 }
           | stmt                   { Stmtlist1 }
stmt      -> USE STRING             { Stmtuse }
           | DEC STRING             { Stmtdec }
           | block                  { Stmtlocal }
```

Figure 3.2: The Grammar Specification for the AG

our AG tool. Below, we explain this particular AG, while at the same time giving an overview of AGs in general and our tool in particular.

3.1.2 An Overview of the AG

Figure 3.2 (page 36) contains the grammar for the language. The AG tool generates a parser for the language, so some parser specific details are also listed here. In particular an entry point for the parser (`program`) and lexer tokens are listed. The Figure also declares the attributes that are used by the AG. The functions for computing the attribute values are given in Figure 3.3 (page 37). The next paragraph gives an overview of the four attributes that are used.

The Four Attributes

The AG uses four attributes: `level`, `locs`, `env` and `code`. The `level` attribute is an integer attribute that represents the depth of the current scope.

```
header <---
open Testtypes
open Testlib ;;

Stmtlocal  block.level  <--- $stmt_p.level + 1 ;;
Main       block.level  <--- 0 ;;

Main       block.env    <--- [] ;;
Block      stmtlist.env <--- add $block_p.level $stmtlist.locs
                                 $block_p.env ;;

Main       program_p.code <---Text ~$block.code~ ;;
Block      block_p.code   <---Text
Enter(~string_of_int $block_p.level~,
      ~string_of_int (List.length $stmtlist.locs)~)
~$stmtlist.code~
Exit(~string_of_int $block_p.level~)
;;

Stmtlist2  stmtlist_p.code <---Text
~$stmt.code~
~$stmtlist.code~ ;;

Stmtlist1  stmtlist_p.code <---Text ~$stmt.code~ ;;
Stmtuse    stmt_p.code     <---Text
Ref ~showintpair (lokup $stmt_p.env $STRING.lex)~ ;;

Stmtdec    stmt_p.code     <---Text ;;
Stmtlocal  stmt_p.code     <---Text ~$block.code~ ;;

Stmtlist2  stmtlist_p.locs <--- $stmt.locs @ $stmtlist.locs ;;
Stmtlist1  stmtlist_p.locs <--- $stmt.locs ;;
Stmtuse    stmt_p.locs     <--- [] ;;
Stmtdec    stmt_p.locs     <--- [$STRING.lex] ;;
Stmtlocal  stmt_p.locs     <--- [] ;;
```

Figure 3.3: The Semantic Functions for the AG

```
Enter(0, 2)
Ref (0, 0)
Ref (0, 1)
Enter(1, 1)
Ref (1, 0)
Ref (0, 0)
Exit(1)
Ref (0, 0)
Exit(0)
```

Figure 3.4: The Translation of the program in Figure 3.1 (page 35)

The list of local variables declared in the current scope is given by `locs`. The `env` attribute is defined recursively and also refers to `level` and `locs`. It represents the current environment, which is a mapping from variable names to the lexical level and offset at which they were declared. Note that the type of `env` is declared in Figure 3.2 (page 36). It is `Testtypes.envir`, which is equivalent to `(string * (int * int)) list` (an association list). When a new scope is introduced, the environment in the new scope is defined by extending the association list. This is done by the OCaml function `add`, which is defined in Figure 3.5 (page 39).

The `code` attribute gives the translation of the language. The translation is to a simple stack-based language. It has the following three commands:

- `Enter(d, l)` Open a new stack frame at depth `d`. The frame should be of size `l`.

- `Exit(d)` Exit the stack frame at depth `d`.

- `ref(d, x)` Access the variable at offset `x` in the stack frame at depth `d`.

For every new scope that is introduced in the source code, a corresponding stack frame is introduced in the translation. The translation of the input in Figure 3.1 (page 35) is given in Figure 3.4 (page 38).

```
open Testtypes

let lokup env x = List.assoc x env ;;

let showintpair (x,y) = Printf.sprintf "(%d, %d)" x y

let add level locs env =
  let rec newvars locs id =
     match locs with
       [] -> []
     | x :: xs -> (x,(level,id)) :: newvars xs (id+1)
  in newvars locs 0 @ env ;;
```

Figure 3.5: The Library Module `testlib.ml`

3.1.3 The Attribute Definition Functions

Figure 3.3 (page 37) gives the rules for computing the attribute values. Each rule is associated with a particular production of the grammar and defines an attribute value for one of the non-terminals that appears in that production. These rules are applied recursively to the syntax tree.

Inherited and Synthesised Attributes

In Figure 3.2 (page 36), the four attributes used by the AG are declared. The declarations specify not only the OCaml type of the attribute, but also whether the attribute is "Inherited" or "Synthesised". The distinction is between attribute values that are passed up the tree (synthesised) and attribute values that are passed down the tree (inherited). The `level` and `env` attributes are inherited, because they are passed down the tree. For example, when a new scope is introduced, an environment is computed for it and passed down to it. The `code` and `locs` attributes are synthesised, because they are of interest to nodes further up the tree.

Inherited attributes can optionally be defined to be copyable. Suppose a node of the tree has a `level` attribute but there are no explicit rules to define the `level` attributes of the node's children. The `level` attribute is copyable, so the children of the node are automatically given the same value of `level` as the parent. This default behaviour can often save a lot of work by

removing the need to write out attribute rules that simply copy the attribute value. Unfortunately there isn't such an obvious default behaviour to give to synthesised attributes. This is because a node of the tree may have several children, which each have different values of the attribute. In this situation it would be unclear which of the attribute values should be copied.

Syntax and Behaviour of the Attribute Rules

Figure 3.3 (page 37) gives the rules for computing the attributes. The composition of each rule is as follows:

`Stmtlocal`	`block.level`	`<---`	`$stmt_p.level + 1`	`;;`
production ID	attribute to be computed		attribute value	

Each attribute rule is associated with a particular production. This is done by the name at the beginning of the rule. These names (which are capitalised by convention) appear in braces after each production in the grammar definition. This can be seen in Figure 3.2 (page 36).

Each rule computes an attribute value for one non-terminal in the production. The value of the attribute is given by an OCaml expression. Attribute values of non-terminals in the production can be accessed inside the OCaml expression by using the notation `$node.attribute`.

Note that when an attribute rule refers to the non-terminal that appears on the left hand side of the production, the name of the non-terminal is extended with `_p`. This is an example of the tool's naming scheme, which is intended to improve readability and avoid ambiguity. A good example of the scheme is given by the production: `stmt -> IF expr THEN stmt ELSE stmt`. Here, the three instances of `stmt` are referred to as `stmt_p`, `stmt_1` and `stmt_2`, respectively. On the other hand, there is only one instance of expr, so it is simply referred to as `expr`.

Order of the Attribute Rules

The attribute rules can appear in any order in the specification file. In Figure 3.3 (page 37) they have been grouped by the attribute that they compute.

```
type envir = (string * (int * int)) list
type locstype = string list
```

Figure 3.6: The Library Module `testlib.ml`

Text Quoting Facilities

As described above, the value of attributes is usually given by OCaml expressions. This can be inconvenient when the attribute consists predominantly of ASCII text. A quoting facility is provided for this situation. This is used to define the `code` attribute. The value of the attribute is delimited by `<---Text` and `;;` rather than `<---` and `;;`. The text that appears here is copied verbatim. An anti-quote mechanism (delimited by the `~` symbol) is provided for splicing in OCaml expressions.

Text quoting may seem a trivial and obvious facility, but it can be a great productivity boost. Rather than viewing it as a mere accessory, Van Wyk [30] considers it to be an example of a Domain Specific Meta Language. He suggests that other such languages could be used for more advanced tasks such as writing type-checkers.

Accessing Code from External Modules

Some of the attribute rules in Figure 3.3 (page 37) use OCaml functions that are loaded from other modules. These modules are loaded by the `header` statement at the top of the file. The code that appears in these modules is shown in Figure 3.5 (page 39) and Figure 3.6 (page 41).

3.2 Current Features of the AG Tool

In this Section we describe the AG tool. The tool is programmed in OCaml. The attribute evaluators generated by the tool are also written in OCaml. The tool is quite lightweight, yet it provides most of the key features of an AG system. As such it should be a good starting point from which to create a prototype IP system.

3.2.1 Positive Features of the AG Tool

OCaml pre-processor

The tool is implemented as a pre-processor. It translates AG specifications such as those given in Figure 3.2 (page 36) and Figure 3.3 (page 37) to OCaml programs. The OCaml programs use Yacc to create the parser for the language. This approach has the advantage that the functions for computing the attribute values are written in OCaml. This is a cheap way of creating a powerful system.

Johnsson [13] has shown that there is a natural way of evaluating AGs in a lazy functional programming language. This method, which uses lazy evaluation to create a demand driven attribute evaluator, can be used to evaluate the most general possible class of AGs. OCaml is not a lazy language, but it allows the simulation of laziness by using a library module called “Lazy”. Therefore the AG tool generates OCaml code which uses the Lazy module to emulate the code suggested by Johnsson.

Error checking

In an AG, not all nodes will be annotated with every attribute. This means that the programmer could make an error by trying to read an attribute from a node that doesn't have that attribute. These errors can be caught by doing a closure check [17]. That is, checking that the set of attribute uses is a subset of the set of attribute definitions.

The AG tool uses the information gained during the closure check to improve the efficiency of the translator. Each node only allocates storage for the attributes that it uses, rather than for every existing attribute.

3.2.2 Negative Features of the AG Tool

The AG tool is still in its infancy. As a result it has some shortcomings:

monolithic The AG tool currently offers very little support for breaking AGs into components. This ability will be crucial in an IP system, so a good module system is needed.

parser oriented The tool only allows one grammar to be defined. This grammar is used for both parsing and for defining attribute values.

This means that in some cases the grammars will contain detail which is necessary for parsing, but irrelevant when defining attribute values. In these cases it would be helpful to distinguish between the parsing grammar and the internal grammar. These are often referred to as the concrete and abstract syntax.

no higher order features The tool does not yet support higher order or attribute coupled AGs. These concepts are explained in Section 3.3 below.

no polymorphic type-constructors It is not currently possible to define polymorphic constructions like lists and maybes. This facility would be very helpful when defining grammars.

Three of these four problems have well-known solutions. Adding these solutions to our system is merely an engineering issue. However the lack of facilities for modularity is a more difficult issue. Below we describe some of the previous work that has been done on making AGs modular. In Section 3.4 (page 47) we illustrate some of the potential difficulties with a detailed example.

3.3 Previous Work on Making AGs Modular

The ability to modularise attribute grammars is a necessity if they are to be used in intentional programming. Otherwise intentions could not be defined as separate entities. The problem of modularising AGs has been tackled before, but the traditional motivation has been slightly different to ours.

AG specifications written with only the primitive AG facilities are often highly redundant. This is because many very similar computations need to be painstakingly written out for every production in the grammar. Both Dueck and Cormack [3] and Kastens and Waite [15] ascribe the problem to a lack of appropriate abstraction facilities. Dueck and Cormack propose the use of patterns as a solution. Kastens and Waite suggest three techniques: remote attribute access, symbol computations and inheritance. We give a brief description of these techniques below.

Compilation is traditionally divided into phases. Between each phase, the internal representation of the program changes with the final representation being the translation. In the framework of attribute grammars, this kind

of modularity can be supported by using attribute coupled grammars [9] or higher order attribute grammars [28]. In an attribute coupled grammar, each phase of the compiler is described by a new AG. The AGs that describe intermediate phases of the compiler return the syntax tree for the next phase as a synthesised attribute on the root node. Higher order AGs are a generalisation of attribute coupled grammars in which attributed trees can be passed around anywhere in the tree, rather than just on the root node.

3.3.1 Pattern AGs

The observation made by Dueck and Cormack [3] is that many attribution rules depend on the structure of the production, but not on the content. As an example, they discuss the process of passing environments around the tree. Compilers often need to pass an environment down the tree while passing new variable definitions up the tree. This operation, for which Jullig and DeRemer [14] coined the phrase "Bucket Brigade", is a left to right traversal of the tree. Dueck and Cormack use pattern notation to describe the operation as follows:

module *env*
  (1) 'goal → A . . .
    A.env = 0;
  (2) A → B . . .
    B.env = A.env;
  (3) A → . . . B C . . .
    C.env = B.def;
module *def*
  (4) A → . . . B
    A.def = B.def;
  (5) A →
    A.def = A.env;

This code assumes that the root node of the tree is called "goal". Variables such as A and B are used to match terminals and non-terminals of the grammar. The pattern ". . ." matches zero or more symbols. In the above definition, env is an inherited attribute and def is a synthesised attribute. The five rules describe how these attributes should be passed around the tree in the absence of a specific rule. This is done by matching the above

patterns against every production of the grammar. If the pattern matches a production, then the attribute computation is added to that production. That is, unless a specific rule exists for that attribute on that production. To create the environment behaviour for a specific AG, the above pattern rules need to be augmented with specific rules that recognise defining occurrences of identifiers and modify the environment accordingly.

Disadvantages of Pattern AGs

As a mechanism for reuse, the patterns of Dueck and Cormack can be a little limited. For example, suppose that an AG needs a second bucket brigade, involving the attributes x and y. Then the code given above needs to be written out a second time with x and y substituted for *env* and *def*, respectively. For proper reuse to be possible, patterns need to be parameterisable.

3.3.2 Remote Attribute Access

Kastens and Waite [15] note that AGs frequently need to transfer an attribute value from one position in the tree to another. To do this, attribute rules that copy the value need to be written out for every stage of the transfer path. Kastens and Waite identify three common kinds of remote access and introduce special notation for them:

1. In a traditional attribute grammar, inherited attributes can only be read from the parent node of a production. Kastens and Waite note that it would often be convenient to read inherited attributes of ancestors of the production. They introduce the notation: `including N.a`. This expression evaluates to the value of the attribute `a` on the most recent ancestor which is a non-terminal of type `N`.

2. A common idiom in attribute grammars is to gather a set of information from all the descendents of a production. Kastens and Waite introduce the notation: `constituents N.a with (t, union, single, null)`. This expression evaluates to a value of type `t`. It is computed by folding over all occurrences of the attribute `a` on descendent nodes of type `N`. The fold operation uses the functions `union`, `single` and `null` to build a value of type `t`.

3. In Section 3.3.1 (page 44), we discussed a common idiom called the "Bucket Brigade". In this idiom, an attribute is passed around via a left-to-right traversal of the tree. The idiom is often used to compute environments. Kastens and Waite introduce the notation: `chainstart N.a = e`. This initialises the attribute `a` on the node `N` in the current production. It also adds default computations to the subtree below `N`. Unless a specific rule is given, these default computations copy the value of the attribute via a left-to-right traversal of the subtree.

The current implementation of our AG tool offers a simplified version of "including". As described in Section 3.1.3 (page 39), Inherited attributes can be defined to be "copyable".

3.3.3 Symbol Computations

In traditional attribute grammars, attribute computations are always associated with productions. Kastens and Waite [15] note that attribute values are sometimes independent of the production that they find themselves in. Suppose for example that `stmt` non-terminals have an attribute `xs`, which is a list. Suppose that we wish to add an attribute `n` to `stmt` which is equal to the length of `xs`. In a conventional attribute grammar, this would involve associating a computation for `n` with every production for `stmt`. A symbol computation would allow us to associate the computation with `stmt`, so it would no longer need to be duplicated.

3.3.4 Inheritance

In ALGOL 68 [29], new scopes can be introduced with three different language constructions: programs, serial clauses and procedures. To simplify the description of the semantics, the designers introduced an imaginary language construction called a "range". The behaviour of a range is 'inherited' by the three scope-introducing language constructions. Kastens and Waite [15] suggest an inheritance model for attribute grammars. In their model, symbols of the grammar can inherit from each other. Symbol computations (see above) that are defined on the base symbol are inherited. If desired, some of these symbol computations can be redefined by the inheriting symbol.

Kastens and Waite only allow symbol computations to be inherited. In other words, attribution rules that are associated with productions cannot

be inherited. This is because it is the symbol that is being inherited, not the production. In contrast, the IP system being developed at Microsoft allows the behaviour of a production to be inherited (via a mechanism called forwarding). This facility has proved to be most useful when the inherited production is defined as a composition. For example in a language that has conditionals and gotos, the "while" construction could be defined as follows:

```
while(C,B)   inherits:   {
                         start:
                                 if (C) {
                                        B
                                        goto start;
                                 }
                         }
```

In the above notation, while(C,B) represents the following production:

```
while → condition stmt
```

The structure on the right is a pretty-printed version of the composition of a number of production rules. IP also allows attribute rules to be overridden in the new production.

3.4 Modular AGs: A Case Study

In this section, we describe a detailed example of a language extension. We discuss the issues involved in describing the extension as a detachable module. We show that the majority of the extension can be neatly modularised. However, a few difficulties do arise. In particular, the code for the procedure environment needs to be modified in a rather non-modular way.

In the example, we add exceptions to a simple imperative language. The code for this example, which was written using our AG tool, is given in Appendix A (page 59). Below we first give a brief description of the language and the extension. Then we discuss the degree of modularity that we managed to achieve in the specification of the extension.

```
let var int fib_zero in
fib_zero := 1 ;
let proc fib (val int n, ref int result)
    if n == 0 then result := fib_zero
              else let var int x in
                   fib(n - 1, x)
                 ; result := n * x
    fi in
let var int x in
fib(10, x)
```

Figure 3.7: A Program for Computing the Fibonacci Function

```
void f5(int p2, int& p1, int& x3)
    {
    if ((p2 == 0))
        p1 = x3;
    else
        {
        int x4;
        {
        f5((p2 - 1), x4, x3);
        p1 = (p2 * x4);
        }
        }
    }
```

```
void main()
    {
    {
    int x3;
    {
    x3 = 1;
    {
    int x6;
    f5(10, x6, x3);
    }
    }
    }
    }
```

Figure 3.8: The C Translation of the Fibonacci Program

3.4.1 The Base Language

The base language is a simple imperative language. Its grammar is given in Appendix A.1.1 (page 60). An example program written in the language is given in Figure 3.7 (page 48). This program computes the tenth Fibonacci number. The C code that the program translates to is given in Figure 3.8 (page 48).

```
let exception divbyzero in
let proc divide (val int a, val int b, ref int result) : divbyzero
    if b == 0
       then raise divbyzero
       else if a < b
               then result := 0
               else let var int x in
                    divide(a-b, b, x)
                  ; result := x + 1
            fi
    fi in

let var int x in
try divide(10,2,x)
with divbyzero -> skip
end
```

Figure 3.9: A Program for Computing Integer Division

3.4.2 The Language Extension

The extension adds exceptions to the base language. This means that the syntax of the language is extended with "try-with" and "raise" constructions. There is also a new kind of declaration for introducing exception names. Finally, procedure declarations must now be annotated with the exceptions that they might raise. The details of these modifications to the grammar are given in Appendix A.2.1 (page 69).

An example of a program that uses exceptions is given in Figure 3.9 (page 49). The C translation is given in Figure 3.10 (page 50). The translation uses a global variable called `exception` of type `exception_enum`. A "raise" in the source language is translated to an assignment to this variable followed by a jump command. The jump command jumps to the nearest enclosing "catch" for the exception or if there is none available it returns from the procedure. Therefore the translation of a procedure call is slightly more complicated than it used to be. After a call to a procedure that might raise an exception, the value of the global variable `exception` is checked. If it is set, then another jump command is executed.

```
enum exception_enum {
    NOEXCEPTION,
    E1
};

exception_enum exception;

void f6(int p4, int p3, int& p2)
    {
    if ((p3 == 0))
        {
        exception = E1;
        return;
        }
    else
        if ((p4 < p3))
            p2 = 0;
        else
            {
            int x5;
            {
            {
            f6((p4 - p3), p3, x5);
            switch (exception) {
            case E1:
                return;
            case NOEXCEPTION:
                break;
            }
            }
            p2 = (x5 + 1);
            }
            }
    }
```

```
void main()
    {
    {
    int x9;
    {
    {
    f6(10, 2, x9);
    switch (exception) {
    case E1:
        goto l8;
    case NOEXCEPTION:
        break;
    }
    }
    goto l7;
l8: ;
    switch (exception) {
    case E1:
        exception = NOEXCEPTION;
        ;
        break;
    }
l7: ;
    }
    }
    }
```

Figure 3.10: The C Translation of the Divide Program

3.4.3 Modularising the Translator

The translator for the base language consists of three parts: the grammar, the attribution rules and the support code, written in OCaml. We have attempted to add exceptions to the language by modularly extending these three parts. In this section we describe an idealised view of how this process should proceed. In the next section we describe the problems that we encountered in practice.

Extending the Grammar

Extending the grammar is done by adding new productions. For example in our extension, we wish to add a new statement for raising an exception. This is done by adding the following production:

stmt → RAISE IDENTIFIER { Raise }

This extension does not alter any existing grammar productions, so in that sense it is modular. However, it will not integrate into the existing translator unless it is accompanied by appropriate attribution rules. This is described below.

Sometimes we may need to alter an existing production. For example in our extension, a procedure definition can be annotated with the list of exceptions that it might raise. The original production was:

decl → PROCDEF IDENTIFIER paramdeflist stmt { ProcDef }

In the modified production, an extra non-terminal (exns) is added for the list of exceptions:

decl → PROCDEF IDENTIFIER paramdeflist exns stmt { ProcDef }

Existing attribution rules ignore the "exns" non-terminal, so this extension does not affect them. In that sense this extension is modular. However, if two separate language extensions try to modify the same production, then a conflict may arise. In this situation extra code may need to be written which describes how the two extensions should interact.

Extending the Attribution Rules

When new grammar productions are added, there are three ways in which the attribution rules need to updated. Firstly, attribution rules need to be defined for the new grammar productions such that they integrate into the existing AG. For example, when we add the “Raise” production (given above) to our language, we need to define “stmt.code” for that production. Secondly, new attributes may need to be defined. For example, our extension needs to define a range of new attributes that implement the semantics of exceptions. These attributes are not only added to the new productions, but also to the existing ones. Thirdly, existing attribution rules may need to be modified. In our example, this happens to a number of the rules for “code”. This is because extra code needs to be generated for implementing the exceptions.

We wish to make the implementation of our language extension as independent as possible from the implementation of the base language. In other words, we wish to avoid the second and third kinds of modifications listed above. This might be possible if the base-language implementation provides an appropriate set of hooks for the extension to use. Finding examples of such hooks is an area that we intend to research.

Extending the Support Code

The support code for the translator is written in OCaml, which has a module system. Therefore new support code can easily be added by adding a new OCaml module. Non-modular extensions of the OCaml code happen when the original code needs to be modified. As we shall describe below, this happened in our translator, because a type definition needed to be changed.

3.4.4 Modularisation Difficulties

There are two areas in which our language extension is distinctly non-modular. The first is code-generation and the second is the procedure environment. The attribution rules which are involved are given in Appendix A.2.2 (page 75).

We needed to modify two attribution rules for the “code” attribute. They are the rules for the “Program” and “ProcCall” productions. The modification in the “Program” production creates a global variable for storing the exception that is currently being raised. The modification in the “ProcCall”

production generates the extra code that is needed for handling exceptions after a procedure call. These modifications are non-modular, because they alter the implementation of the original translator. Unfortunately there is no obvious solution to this problem. The modifications need to modify the code generation process, so they are inherently non-modular.

In contrast, the problem with the procedure environment is induced by our specification language rather than being an inherent problem. Due to the inclusion of exceptions we now wish to carry an extra piece of information in the procedure environment: the list of exceptions that might be thrown by that procedure. It is not easy to modify the original procedure environment. In the base language its type is defined to be `procinfo Stringmap.t`, where the type `procinfo` is:

```
type procinfo = (Internalname.t * (typeval * paramkind) list
                * Internalname.t list)
```

In the extended version of the language, we need to redefine `procinfo`:

```
type procinfo = (Internalname.t * (typeval * paramkind) list
                * Internalname.t list * exnuses)
```

This is a distinctly non-modular change, because it affects every piece of code that interacts with the environment. This is a shame, because conceptually the operation that we are trying to perform is quite modular: we are merely trying to pass a new piece of information around according to the scope rules for procedures. By packaging all such information into one tuple we are making an efficiency saving and reusing the code for passing environments around. It would be nice if we could retain the efficiency saving and code reuse, but also have a modular description.

3.5 Further Work: A Module System for Attribute Grammars

In Section 3.4 above we described a case study in which we added an extension to a language. Although our description of the extension was reasonably modular, it is clear that a better module system is required for our AG specification language. In our continuing research, we intend to design an appropriate module system. In this section, we shall present a few ideas for the design.

3.5.1 Module Interfaces

A module is a component that can be described by an interface. The interface should be a concise description of the "meaning" of the module. When we design our module system, we need to decide what information needs to be contained in the interface for a module.

In our case study, we saw that our extension added new productions to the grammar. It defined attribute values on these new productions, but also on productions of the base language. It also read attribute values defined by the base language. These shared productions, non-terminals and attributes are all points where a connection is formed between the extension and the base language. Therefore, they should be included in the interface.

Glue

In our case study, we also saw that attribute grammar modules may not always fit together seamlessly. For example, in our extension we needed to modify the production rule "ProcDef" of the base-language. In cases like this where the incompatibilities between two modules are only 'superficial', a simple glue-language could be used to guide the composition. At present the 'superficial' incompatibilities that we foresee are the following:

- The abstract syntax used in the two modules is incompatible, but essentially very similar. An example of this is the problem with the "ProcDef" production. In this case the glue-code could specify a new abstract syntax and mappings from this new abstract syntax to the abstract syntax of the two modules. This would allow both modules to operate on the new abstract syntax.

- The two modules might use different names for the same object. For example the first module might refer to the non-terminal "stmt", which is called "stat" in the second module. The glue-code could apply a renaming to resolve this.

- The two modules might use a different data-representation for the same attribute. For example, both modules might refer to the environment attribute, but one represents it as an association list, whereas the other uses a hash-table. The glue-code could provide two functions for converting association-lists to hash-tables and vice-versa.

We believe that it is better to provide a glue-language than to allow modules to modify the internals of other modules. Otherwise, modules which perform modifications would have very complicated interfaces.

3.5.2 Improved Abstraction Facilities

In the case study, a modularity problem occurred when we wanted to pass extra information around in the procedure environment. Due to a lack of abstraction facilities, this involved modifying the code for the base language. We suggest that an abstraction facility is required that allows us to describe the procedure environment in a polymorphic way. That is, the code for the environment should be independent of the data that it passes around. It would then be easier to modify the data contained in the environment. In Section 3.3.1 (page 44), we saw that patterns can be used to achieve this kind of effect. However, we noted that patterns lack certain kinds of parameterisation. We believe that there may be a better solution that is based on the use of polymorphism and higher order functions in a functional language. Such a solution would also be simple to implement, because we are currently using OCaml to evaluate our attribute grammars.

Chapter 4

Conclusion

In this dissertation, we have suggested that programming languages could be created by composing reusable language components. In particular, Domain Adapted Languages could be created by adding a Domain Specific Language Extension to a general purpose base language. Such reuse could make the process of designing new languages simpler and quicker.

We wish to research the feasibility of language design reuse. We intend to do this by conducting three related pieces of work. Firstly we need to find a formalism for describing language components. In Chapter 3 (page 34), we showed that with an appropriate module system, attribute grammars will be suitable. Secondly, we need to test our formalism by building a new programming language. We hope to demonstrate that our formalism allows this to be done in a modular manner. In Chapter 2 (page 15), we described a design for a language extension for Graphical User Interface programming. This is the example that we shall use. Thirdly, we need to test our new programming language by writing programs. We plan to write a text-editor with similar functionality to Wordpad, an editor that is shipped with the Microsoft Windows Operating System.

4.1 Research Goals

As described above, our research plan consists of three main components:

- Design an attribute grammar system.
- Design a GUI language extension.

- Write programs with the GUI language.

Our work on attribute grammars will focus on creating a module system that will allow us to describe language components as separate modules. We also need to design a "glue" language that will allow us to compose language components, even when there are minor conflicts between the components. Our experiment with GUI programming is intended to focus on describing the extension as a reusable component. We would like to write the extension in such a way that it can be added to more than one base language. Therefore, our focus is not on finding new GUI programming techniques.

The schedule that we aim to follow is given in Table 4.1 (page 58).

4.1.1 Secondary Goals

In addition to the three main research goals listed above, there are some other smaller topics that we would like to address. Some of these topics are suggestions rather than definite plans. We may change this list in the future.

- We would like to research the similarity between attribute grammars and the framework being used in Microsoft's IP project. We intend to do this by modeling the Microsoft framework as an extension version of attribute grammars. This work will be done in collaboration with E. Van Wyk.

- We would like to add domain specific error checking to our GUI language. As we said in Section 1.2 (page 8), domain specific error checking is an important motivation for introducing new language constructions. We would like to implement an analysis tool that verifies that there are no race conditions in the user interface of a program. (A race condition might occur if there is a circular sequence of widget messages.)

- We would like to add our GUI language extension to more than one base language. Initially, we shall add the extension to a "toy" imperative language. If possible, we would also like to add the extension to a "real" programming language. For this, we will need an attribute grammar specification of the programming language. Unfortunately, we do not know of any such specifications that are publically available.

Dates		*Task*
2000	*March*	Write paper on modeling Microsoft's IP framework with attribute grammars.
	Apr – May	Work on new attribute grammar tool.
	June – July	Implement GUI language extension.
	Aug – Sept	Write editor program with GUI language.
	October	Write paper on the attribute grammar module system.
	Nov – Dec	Revise the attribute grammar system, based on our experiences with the GUI language.
2001	*Jan – Feb*	Implement new demos with the revised attribute grammar tool.
	March	Write paper on improved attribute grammar tool.
	Apr – Sept	Write thesis.

Table 4.1: Research Schedule

Appendix A

Adding Exceptions to an Imperative Language

This appendix contains the code for two translators. The first translator translates a small imperative language to code written in C. The second translator translates an extended version of the language that includes exceptions. The code for these translators is mostly written using our AG tool, which is described in Chapter 3 (page 34). The remaining code consists of a number of utility modules, written in plain OCaml.

We have divided our description of the attribution rules into 'aspects'. Although our AG tool does not currently provide any support for aspect oriented modules, we have given an interface for each aspect. The interface consists of three elements:

1. Whether the attribute is inherited or synthesised.

2. The type of the attribute

3. The non-terminals on which the attribute is 'visible'. That is, the non-terminals on which the attribute value is allowed to be read by other modules.

Some of our aspects use internal attributes. These are attributes which are only visible within that aspect. Such attributes are not listed in the interface of the module.

A.1 The Base Language

The base language is a simple imperative language. It has procedures (but not functions), that can be nested. It has two datatypes: integers and multi-dimensional arrays of integers. A number of basic operations on these datatypes are provided. At present, the compiler presented here does not type-check the code. Such functionality could easily be added though. Another implementation flaw is that arrays which are supposed to be passed by value are actually passed by reference. Again, the necessary code could easily be added.

A.1.1 The Grammar

program	→	stmt EOF	{ Program }
stmt	→	LET decl IN stmt	{ Declaration }
	|	stmt SEMICOLON stmt	{ Composition }
	|	SKIP	{ Skip }
	|	IF expr THEN stmt ELSE stmt FI	{ Conditional }
	|	WHILE expr DO stmt OD	{ While }
	|	expr ASSIGN expr	{ Assign }
	|	IDENTIFIER paramlist	{ ProcCall }
expr	→	IDENTIFIER	{ Variable }
	|	INTCONST	{ IntConst }
	|	expr LBRACKET expr RBRACKET	{ ArrayAccess }
	|	expr PLUS expr	{ Plus }
	|	expr MINUS expr	{ Minus }
	|	expr TIMES expr	{ Times }
	|	expr AND expr	{ And }
	|	expr OR expr	{ Or }
	|	expr LESSTHAN expr	{ LessThan }
	|	expr EQUALS expr	{ Equals }
	|	LPAREN expr RPAREN	{ ExprGroup }
decl	→	PROCDEF IDENTIFIER paramdeflist stmt	{ ProcDef }
	|	VARDEF typeval IDENTIFIER	{ VarDef }
typeval	→	INT	{ IntType }
	|	ARRAY INTCONST OF typeval	{ ArrayType }
paramdeflist	→	LPAREN paramdefs RPAREN	{ Paramdeflist }
	|	LPAREN RPAREN	{ Paramdeflist0 }

paramdefs	→	paramdef COMMA paramdefs	{ Paramdefs }
	|	paramdef	{ Paramdefs1 }
paramdef	→	paramtype typeval IDENTIFIER	{ Paramdef }
paramtype	→	VAL	{ ParamVal }
	|	REF	{ ParamRef }
paramlist	→	LPAREN params RPAREN	{ Paramlist }
	|	LPAREN RPAREN	{ Paramlist0 }
params	→	expr COMMA params	{ Params }
	|	expr	{ Params1 }

A.1.2 `Imperativelib.mli`

This file is the interface file for `Imperativelib.ml`, which is an OCaml module which contains a number of functions that are used by the translator. It also defines a number of types.

```
open Kevutils

exception AnalysisError of string

type typeval = IntType | ArrayType of int * typeval
type paramkind = Val | Ref
type paramdef = (string * Internalname.t * typeval * paramkind)
type varinfo = (Internalname.t * typeval)
type paraminfo = (Internalname.t * (typeval * paramkind))
type procinfo = (Internalname.t * (typeval * paramkind) list * Internalname.t list)

module VarInfoSet : (Set.S with type elt = varinfo)

type varuses = VarInfoSet.t
type varenvir = varinfo Stringmap.t
type procenvir = procinfo Stringmap.t
type varusesclosure = varuses -> varuses
type codeclosure = string -> string

val varlokup : varenvir -> string -> varinfo
val proclokup : procenvir -> string -> procinfo
val paraminfo_of_paramdef : paramdef -> paraminfo
val add_paramdefs_to_varenvir : paramdef list -> varenvir -> varenvir
val varuses_of_paramdefs : paramdef list -> varuses
val convert_to_ctype : Internalname.t -> typeval -> paramkind ->
                       C.typeval * C.variable
```

A.1.3 `Imperativelib.ml`

This file is the implementation of the interface listed above.

```
open Kevutils
```

```
exception AnalysisError of string

type typeval = IntType | ArrayType of int * typeval
type paramkind = Val | Ref
type paramdef = (string * Internalname.t * typeval * paramkind)
type varinfo = (Internalname.t * typeval)
type paraminfo = (Internalname.t * (typeval * paramkind))
type procinfo = (Internalname.t * (typeval * paramkind) list * Internalname.t list)
type declkind = VarDecl of varinfo | ProcDecl of (Internalname.t * paraminfo list)

module VarInfo =
  struct
    type t = varinfo
    let compare = fun (name1,_) (name2,_) -> Internalname.compare name1 name2
  end

module VarInfoSet = Set.Make(VarInfo)

type varuses = VarInfoSet.t
type varenvir = varinfo Stringmap.t
type procenvir = procinfo Stringmap.t
type varusesclosure = varuses -> varuses
type codeclosure = string -> string

let varlokup env x =
    try Stringmap.find x env
    with Not_found ->
         let env_elements = Stringmap.keys env in
         raise (AnalysisError("The variable \"" ^ x ^ "\" does not exist.\n"
                  ^ "The current variable environment is"
                  ^ (stringconcat ": " ", " "." "empty." env_elements)))

let proclokup env x =
    try Stringmap.find x env
    with Not_found ->
         let env_elements = Stringmap.keys env in
         raise (AnalysisError("The procedure \"" ^ x ^ "\" does not exist.\n"
                  ^ "The current procedure environment is"
                  ^ (stringconcat ": " ", " "." "empty." env_elements)))

let paraminfo_of_paramdef = fun (name, newname, typeval, valres) ->
    (newname, (typeval, valres))

let varinfo_of_paramdef = fun (name, newname, typeval, valres) ->
    (newname, typeval)

let add_paramdefs_to_varenvir defs env =
    let add map = fun (name, newname, typeval, valres) ->
        Stringmap.add name (newname, typeval) map in
    List.fold_left add env defs

let varuses_of_paramdefs defs =
    let add set def = VarInfoSet.add (varinfo_of_paramdef def) set in
    List.fold_left add VarInfoSet.empty defs

(* In our imperative language, type declarations are of the form:
```

```
   var array 10 of int x. In C this becomes: int x[10]. In other words,
   the array information needs to be moved onto the variable name.
   This function does the conversion. Its output uses the C
   library to represent the C. *)
let convert_to_ctype name typeval valres =
    let rec mkvar = function
          IntType -> C.Varname(Internalname.mkstring name)
        | ArrayType(n, subtype) -> C.Array(n, mkvar subtype) in
    let cvar = mkvar typeval in
    let ctype = match typeval with
                  IntType -> (match valres with
                                 Val -> C.Int
                               | Ref -> C.Ref(C.Int))
                | ArrayType(_,_) -> C.Int in
    (ctype, cvar)
```

A.1.4 Internalname.mli

Our translator frequently needs to generate unique names for internal variables and labels. This module introduces a new type for this purpose. The `create` function generates a new unique name. The `mkstring` function generates a textual representation of the unique name. The string that was passed as a parameter to the `create` function prefixes this textual representation. This feature can be used to make the unique names slightly more readable. The `compare` function makes this module an instance of `OrderedType`. This makes it compatible with the `Set` and `Map` modules in the OCaml standard library.

```
type t

val create : string -> t
val mkstring : t -> string
val compare : t -> t -> int
```

A.1.5 Internalname.ml

This module implements the above interface. A static counter is used to generate unique numbers.

```
type t = InternalName of string * int

(* The following are used to generate unique function names *)
let unique_counter = ref 0
let get_unique_int () = incr unique_counter ; !unique_counter

let create initstr = InternalName(initstr, get_unique_int ())

let mkstring = fun (InternalName(initstr, n)) ->
```

```
    (initstr ^ (string_of_int n))

let compare = fun (InternalName(_, m)) (InternalName(_, n)) -> m - n
```

A.1.6 `compile.ml`

This module is the top level of the translator. It calls the attribute grammar as a subroutine and prints the results.

```
open Imperative_parser
open Imperativelib
open Lexutils

let _ =
    initialise_lexer ()
  ; let lexbuf = Lexing.from_channel stdin in
    try
      let result = Imperative_parser.start_program Lexer.token lexbuf in
      print_string (C.show (Lazy.force result))
    with Parsing.Parse_error -> ( match get_most_recent_position () with
                                       (linenum,charnum) ->
                                  Printf.eprintf " on line %d in column %d\n"
                                      linenum charnum )
       | AnalysisError(str) -> prerr_string ("Analysis error: " ^ str ^ "\n")
```

A.1.7 Attribution Rules

Here we list the attribution rules for the attribute grammar. We have grouped the rules by aspect. We give an interface for each aspect as described on page 59.

The `proccode` attribute

> *synthesised*
>
> *type:* `C.declaration list`
>
> *visible on:* stmt

The language allows nested procedures to be defined. We are translating to C, which does not have nested procedures. Therefore nested procedures need to be lifted. The `proccode` attribute contains the list of procedures that have been lifted out of a statement.

```
Declaration     stmt_p.proccode <--- $decl.proccode @ $stmt.proccode ;;
Composition     stmt_p.proccode <--- $stmt_1.proccode @ $stmt_2.proccode ;;
Skip            stmt_p.proccode <--- [] ;;
```

```
Conditional     stmt_p.proccode <--- $stmt_1.proccode @ $stmt_2.proccode ;;
While           stmt_p.proccode <--- $stmt.proccode ;;
Assign          stmt_p.proccode <--- [] ;;
ProcCall        stmt_p.proccode <--- [] ;;
VarDef          decl_p.proccode <--- [] ;;
ProcDef         decl_p.proccode <---
   let add_Ref = (fun (name, typeval) -> (name, (typeval, Ref))) in
   let myuses = VarInfoSet.diff
                  $stmt.varuses
                  (varuses_of_paramdefs $paramdeflist.paramdefs) in
   let extra_params = List.map add_Ref (VarInfoSet.elements myuses) in
   let paramlist = List.map paraminfo_of_paramdef $paramdeflist.paramdefs in
   let params = paramlist @ extra_params in
   let code_of_param = fun (name, (typeval, valres)) ->
       match convert_to_ctype name typeval valres with (ctype, cvar) ->
       C.Paramdef(ctype, cvar) in
   let params_code = List.map code_of_param params in
   let thisdef =
       C.Function(C.Void, Internalname.mkstring $decl_p.internalname,
                    params_code, [$stmt.code])
   in
   $stmt.proccode @ [thisdef] ;;
```

The `code` attribute

> *synthesised*
>
> *type:* `C.program` on program, `C.stmt` on stmt
>
> *visible on:* program, stmt

This attribute contains the translation of the program. The C code that the program is translated to is represented in tree form. This tree format is defined in the C module. This attribute has a different type depending on the non-terminal type that it is associated with. For example on *program* it has type `C.program` and on *stmt* it has type `C.stmt`. The lifted procedures that are stored in the `proccode` attribute are added to the code in the `Program` production.

Rather than a `code` attribute, the `decl` non-terminal has a `codeclosure` attribute. The use of closures allows greater flexibility. This is needed in the VarDef production.

```
Program     program_p.code     <---
   $stmt.proccode @ [C.Function(C.Void, "main", [], [$stmt.code])] ;;

Declaration stmt_p.code        <--- $decl.codeclosure $stmt.code ;;
ProcDef     decl_p.codeclosure <--- id ;;
VarDef      decl_p.codeclosure <---
   let name = $decl_p.internalname in
   match convert_to_ctype name $typeval.typeval Val with (ctype, cvar) ->
```

```
   fun stmtcode -> (C.Block(C.Decl(ctype, [cvar]) :: [stmtcode])) ;;

Composition stmt_p.code       <--- C.Block($stmt_1.code::[$stmt_2.code]) ;;
Skip        stmt_p.code       <--- C.Skip ;;
Conditional stmt_p.code       <--- C.IfThenElse($expr.code, $stmt_1.code, $stmt_2.code) ;;
While       stmt_p.code       <--- C.While($expr.code, $stmt.code) ;;
Assign      stmt_p.code       <--- C.Assign($expr_1.code, $expr_2.code) ;;
ProcCall    stmt_p.code       <---
   match proclokup $stmt_p.procenvir $IDENTIFIER.lex with
     (newname, _, extraparams) ->
   let convert_extraparams = List.map (fun name -> C.VarUse(Internalname.mkstring name)) in
   C.ProcCall(Internalname.mkstring newname,
                 $paramlist.code @ convert_extraparams extraparams) ;;

Variable    expr_p.code       <--- C.VarUse(
   Internalname.mkstring (fst (varlokup $expr_p.varenvir $IDENTIFIER.lex))) ;;

IntConst    expr_p.code       <--- C.IntConst($INTCONST.lex) ;;
ArrayAccess expr_p.code       <--- C.ArrayAccess($expr_1.code, $expr_2.code) ;;
ExprGroup   expr_p.code       <--- $expr.code ;;
Plus        expr_p.code       <--- C.Plus($expr_1.code, $expr_2.code) ;;
Minus       expr_p.code       <--- C.Minus($expr_1.code, $expr_2.code) ;;
Times       expr_p.code       <--- C.Times($expr_1.code, $expr_2.code) ;;
And         expr_p.code       <--- C.And($expr_1.code, $expr_2.code) ;;
Or          expr_p.code       <--- C.Or($expr_1.code, $expr_2.code) ;;
LessThan    expr_p.code       <--- C.LessThan($expr_1.code, $expr_2.code) ;;
Equals      expr_p.code       <--- C.Equals($expr_1.code, $expr_2.code) ;;
Paramlist   paramlist_p.code  <--- $params.code ;;
Paramlist0  paramlist_p.code  <--- [] ;;
Params      params_p.code     <--- $expr.code :: $params.code ;;
Params1     params_p.code     <--- [$expr.code] ;;
```

The `procenvir` attribute

inherited (copyable)

type: `procinfo Stringmap.t`

visible on: <all>

This attribute is the environment for procedures. It is a mapping from identifiers to `procinfo`. The type `procinfo` is a tuple containing the internal name of the procedure, the types of its parameters and the list of free variables in its body. (In the C translation the free variables need to be passed as additional parameters, because C does not support nested procedures.)

```
Program         stmt.procenvir  <--- Stringmap.empty ;;
Declaration     stmt.procenvir  <--- Stringmap.merge $decl.procdefs $stmt_p.procenvir ;;
Declaration     decl.procenvir  <--- Stringmap.merge $decl.procdefs $stmt_p.procenvir ;;
VarDef          decl_p.procdefs <--- Stringmap.empty ;;
ProcDef         decl_p.procdefs <---
   let myuses = VarInfoSet.diff $stmt.varuses
```

```
                              (varuses_of_paramdefs $paramdeflist.paramdefs) in
   Stringmap.singleton $IDENTIFIER.lex
                       ($decl_p.internalname,
                        List.map (cp snd paraminfo_of_paramdef) $paramdeflist.paramdefs,
                        List.map fst (VarInfoSet.elements myuses)) ;;
```

The `varenvir` attribute

inherited (copyable)

type: `varinfo Stringmap.t`

visible on: <all>

This attribute is the environment for variables. It is a mapping from identifiers to `varinfo`. The type `varinfo` is a tuple containing the internal name of the variable and its type.

```
Program         stmt.varenvir   <--- Stringmap.empty ;;
Declaration     stmt.varenvir   <--- Stringmap.merge $decl.vardefs $stmt_p.varenvir ;;
ProcDef         stmt.varenvir   <--- add_paramdefs_to_varenvir $paramdeflist.paramdefs
                                                               $decl_p.varenvir ;;
ProcDef         decl_p.vardefs  <--- Stringmap.empty ;;
VarDef          decl_p.vardefs  <--- Stringmap.singleton
                                        $IDENTIFIER.lex
                                        ($decl_p.internalname, $typeval.typeval) ;;
```

The `varuses` attribute

synthesised

type: `VarInfoSet.t`

visible on: stmt

This attribute is used to collect the list of variable uses in a statement. It is used to determine the list of free variable occurrences in a procedure body. This information is needed for the translation to C, because C does not have nested procedures.

```
Declaration stmt_p.varuses <---
   let uses = $stmt.varuses in
   let closure = $decl.varusesclosure in
   closure uses ;;

ProcDef     decl_p.varusesclosure <---
   let myuses = VarInfoSet.diff $stmt.varuses
                              (varuses_of_paramdefs $paramdeflist.paramdefs) in
   fun curruses -> VarInfoSet.union myuses curruses ;;
```

```
VarDef      decl_p.varusesclosure <---
   let mydef = ($decl_p.internalname, $typeval.typeval) in
   fun curruses -> VarInfoSet.remove mydef curruses ;;

Composition stmt_p.varuses      <--- VarInfoSet.union $stmt_1.varuses $stmt_2.varuses ;;
Skip        stmt_p.varuses      <--- VarInfoSet.empty ;;
Conditional stmt_p.varuses      <--- VarInfoSet.union $expr.varuses
                                       (VarInfoSet.union $stmt_1.varuses $stmt_2.varuses) ;;

While       stmt_p.varuses      <--- VarInfoSet.union $expr.varuses $stmt.varuses ;;
Assign      stmt_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
ProcCall    stmt_p.varuses      <--- $paramlist.varuses ;;
Variable    expr_p.varuses      <--- VarInfoSet.singleton
                                         (varlokup $expr_p.varenvir $IDENTIFIER.lex) ;;

IntConst    expr_p.varuses      <--- VarInfoSet.empty ;;
ArrayAccess expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
Plus        expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
Minus       expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
Times       expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
And         expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
Or          expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
LessThan    expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
Equals      expr_p.varuses      <--- VarInfoSet.union $expr_1.varuses $expr_2.varuses ;;
ExprGroup   expr_p.varuses      <--- $expr.varuses ;;
Paramlist   paramlist_p.varuses <--- $params.varuses ;;
Paramlist0  paramlist_p.varuses <--- VarInfoSet.empty ;;
Params      params_p.varuses    <--- VarInfoSet.union $expr.varuses $params.varuses ;;
Params1     params_p.varuses    <--- $expr.varuses ;;
```

The `internalname` attribute

> *inherited*
>
> *type:* `Internalname.t`
>
> *visible on:* decl

During the translation variables and procedures are renamed to ensure uniqueness. The `Internalname` module is used to generate unique names.

```
ProcDef     decl_p.internalnameprefix <--- "f" ;;
VarDef      decl_p.internalnameprefix <--- "x" ;;
Declaration decl.internalname         <--- Internalname.create $decl.internalnameprefix ;;
```

The `typeval` attribute

> *synthesised*
>
> *type:* `typeval`
>
> *visible on:* typeval

This attribute gives the internal representation of a type.

```
IntType   typeval_p.typeval <--- IntType ;;
ArrayType typeval_p.typeval <--- ArrayType($INTCONST.lex, $typeval.typeval) ;;
```

The `paramdefs` attribute

> *synthesised*
>
> *type:* `paramdef list`
>
> *visible on:* paramdeflist

This attribute contains information about the parameters of a procedure. The type `paramdef` is a tuple containing the name of the parameter, its internal name, its type and its passing convention.

```
Paramdeflist  paramdeflist_p.paramdefs <--- $paramdefs.paramdefs ;;
Paramdeflist0 paramdeflist_p.paramdefs <--- [] ;;
Paramdefs     paramdefs_p.paramdefs    <--- $paramdef.paramdefs
                                              :: $paramdefs.paramdefs ;;
Paramdefs1    paramdefs_p.paramdefs    <--- [$paramdef.paramdefs] ;;
Paramdef      paramdef_p.paramdefs     <---
   ($IDENTIFIER.lex, Internalname.create "p", $typeval.typeval, $paramtype.paramkind) ;;
ParamVal      paramtype_p.paramkind    <--- Val ;;
ParamRef      paramtype_p.paramkind    <--- Ref ;;
```

A.2 The Extension

Adding the language extension for exceptions involves modifying both the grammar and the attribution rules. In most cases these modifications can be made by simply adding extra rules to the original specification. However there are a few exceptions, which are noted in the descriptions below.

A.2.1 The Grammar Extensions

To add exceptions to the language, we need to add a number of new productions to the grammar. In particular "try-with" and "raise" constructions are added. A new kind of declaration is also added for declaring new exceptions.

One of the productions below is a modification rather than an addition. The production "ProcDef" was also present in the original grammar, but it has been modified here. It contains a new non-terminal called "exns", which is the list of exceptions that the procedure raises. This list must be explicitly declared by the programmer.

stmt	→	TRY stmt WITH catches END	{ Try }
	|	RAISE IDENTIFIER	{ Raise }
catches	→	catch ALT catches	{ Catches }
	|	catch	{ Catches1 }
catch	→	IDENTIFIER ARROW stmt	{ Catch }
decl	→	PROCDEF IDENTIFIER paramdeflist exns stmt	{ ProcDef }
	|	EXCEPTION IDENTIFIER	{ ExnDef }
exns	→	COLON exnlist	{ Exns }
	|		{ NoExns }
exnlist	→	exn COMMA exnlist	{ ExnList }
	|	exn	{ ExnList1 }
exn	→	IDENTIFIER	{ Exn }

A.2.2 Attribution Rules

To enable the language extension, new attribute rules need to be added and certain old attribute rules need to be redefined. These changes are listed below.

Definitions for existing attributes

We have introduced a number of new productions in previously established contexts. Attribute rules need to be defined to make these productions compatible with their context. For example the "Try" production needs to define a rule for the attribute "varuses" on the "stmt" non-terminal. Below, we list all the new rules of this kind except those defining the `code` attribute. The definitions for the `code` attribute are listed later.

```
ExnDef   decl_p.procdefs           <--- Stringmap.empty ;;
ExnDef   decl_p.vardefs            <--- Stringmap.empty ;;
ExnDef   decl_p.codeclosure        <--- id ;;
ExnDef   decl_p.proccode           <--- [] ;;
Try      stmt_p.proccode           <--- $stmt.proccode @ $catches.proccode ;;
Raise    stmt_p.proccode           <--- [] ;;
Catches  catches_p.proccode        <--- $catch.proccode @ $catches.proccode ;;
Catches1 catches_p.proccode        <--- $catch.proccode ;;
Catch    catch_p.proccode          <--- $stmt.proccode ;;
ExnDef   decl_p.varusesclosure     <--- id ;;
Try      stmt_p.varuses            <--- VarInfoSet.union $stmt.varuses $catches.varuses ;;
Raise    stmt_p.varuses            <--- VarInfoSet.empty ;;
Catches  catches_p.varuses         <--- VarInfoSet.union $catch.varuses $catches.varuses ;;
Catches1 catches_p.varuses         <--- $catch.varuses ;;
Catch    catch_p.varuses           <--- $stmt.varuses ;;
ExnDef   decl_p.internalnameprefix <--- "E" ;;
```

The `exnenvir` attribute

inherited (copyable)

type: `Internalname.t Stringmap.t`

visible on: <all>

The base language has an environment for variables and procedures. Here we introduce a third for exceptions.

```
Program     stmt.exnenvir  <--- Stringmap.empty ;;
Declaration stmt.exnenvir  <--- Stringmap.merge $decl.exndefs $stmt_p.exnenvir ;;
ProcDef     decl_p.exndefs <--- Stringmap.empty ;;
VarDef      decl_p.exndefs <--- Stringmap.empty ;;
ExnDef      decl_p.exndefs <--- Stringmap.singleton $IDENTIFIER.lex $decl_p.internalname ;;
```

The `exncontext` attribute

inherited (copyable)

type: `Internalname.t InternalnameMap.t`

visible on: <all>

A "try-with" construction introduces labelled blocks of code that should be jumped to when an exception occurs. The `exncontext` attribute is an environment mapping exception names to those labels.

The `goto` statement in C only allows jumps within the same procedure. Therefore `raise` commands that appear within a subroutine cannot be translated to a `goto`. Instead they are translated to `return` commands. This is why the ProcDef production passes an empty environment to the procedure body.

```
Try      stmt.exncontext <---
   let labelname = $catches.jumplabel in
   let add_to_env env (exn_name, exn_id) = InternalnameMap.add exn_id labelname env in
   List.fold_left add_to_env $stmt_p.exncontext $catches.caughtexns ;;
Try      catches.exncontext  <--- $stmt_p.exncontext ;;
Program  stmt.exncontext     <--- InternalnameMap.empty ;;
ProcDef  stmt.exncontext     <--- InternalnameMap.empty ;;
Catches  catches_p.jumplabel <--- Internalname.create "l" ;;
Catches1 catches_p.jumplabel <--- Internalname.create "l" ;;
```

The `exnalldefs` attribute

synthesised

type: `Internalname.t list`

visible on: stmt

This attribute collects every exception definition that occurs in the program. This is because the C translation defines the list of all exceptions as a single global `enum`.

```
Declaration stmt_p.exnalldefs    <--- (Stringmap.data $decl.exndefs) @ $stmt.exnalldefs ;;
Composition stmt_p.exnalldefs    <--- $stmt_1.exnalldefs @ $stmt_2.exnalldefs ;;
Skip        stmt_p.exnalldefs    <--- [] ;;
Conditional stmt_p.exnalldefs    <--- $stmt_1.exnalldefs @ $stmt_2.exnalldefs ;;
While       stmt_p.exnalldefs    <--- $stmt.exnalldefs ;;
Assign      stmt_p.exnalldefs    <--- [] ;;
ProcCall    stmt_p.exnalldefs    <--- [] ;;
Try         stmt_p.exnalldefs    <--- $stmt.exnalldefs @ $catches.exnalldefs ;;
Raise       stmt_p.exnalldefs    <--- [] ;;
Catches     catches_p.exnalldefs <--- $catch.exnalldefs @ $catches.exnalldefs ;;
Catches1    catches_p.exnalldefs <--- $catch.exnalldefs ;;
Catch       catch_p.exnalldefs   <--- $stmt.exnalldefs ;;
```

The `exnerrors` attribute

synthesised

type: `string list`

visible on: program

This attribute collects the list of incorrect exception uses. There are two kinds of error. Firstly the programmer might have raised an exception in a procedure, but forgotten to add that exception to the signature of the procedure. Secondly the programmer might have forgotten to catch an exception. This second error is spotted when a raised exception can escape the scope in which it is defined.

```
Program     program_p.exnerrors <--- $stmt.exnerrors ;;
Declaration stmt_p.exnerrors    <---
   let uses = $stmt.exnuses in
   let defs = Stringmap.assoclist $decl.exndefs in
   let uncaught = intersect defs uses in
   let mkmessage = fun (name, id) ->
       "The exception \"" ^ name ^ "\" may be raised and not caught." in
   (List.map mkmessage uncaught) @ $decl.exnerrors @ $stmt.exnerrors ;;
```

```
ProcDef     decl_p.exnerrors    <---
   let undeclared_exns = list_diff $stmt.exnuses $exns.procexns in
   let procname = $IDENTIFIER.lex in
   let mkmessage = fun (name, id) ->
       "The function \"" ^ procname ^ "\" does not declare the "
     ^ "exception \"" ^ name ^ "\" which may be raised during its execution." in
   (List.map mkmessage undeclared_exns) @ $stmt.exnerrors ;;

VarDef      decl_p.exnerrors    <--- [] ;;
ExnDef      decl_p.exnerrors    <--- [] ;;
Composition stmt_p.exnerrors    <--- $stmt_1.exnerrors @ $stmt_2.exnerrors ;;
Skip        stmt_p.exnerrors    <--- [] ;;
Conditional stmt_p.exnerrors    <--- $stmt_1.exnerrors @ $stmt_2.exnerrors ;;
While       stmt_p.exnerrors    <--- $stmt.exnerrors ;;
Assign      stmt_p.exnerrors    <--- [] ;;
ProcCall    stmt_p.exnerrors    <--- [] ;;
Try         stmt_p.exnerrors    <--- $stmt.exnerrors @ $catches.exnerrors ;;
Raise       stmt_p.exnerrors    <--- [] ;;
Catches     catches_p.exnerrors <--- $catch.exnerrors @ $catches.exnerrors ;;
Catches1    catches_p.exnerrors <--- $catch.exnerrors ;;
Catch       catch_p.exnerrors   <--- $stmt.exnerrors ;;
```

The `exnuses` attribute

synthesised

type: `(string * Internalname.t) list`

visible on: stmt

This attribute enumerates the exceptions that might be raised by a particular statement. It is used to calculate the `exnerrors` attribute.

```
Declaration stmt_p.exnuses    <--- $stmt.exnuses ;;
Composition stmt_p.exnuses    <--- $stmt_1.exnuses @ $stmt_2.exnuses ;;
Skip        stmt_p.exnuses    <--- [] ;;
Conditional stmt_p.exnuses    <--- $stmt_1.exnuses @ $stmt_2.exnuses ;;
While       stmt_p.exnuses    <--- $stmt.exnuses ;;
Assign      stmt_p.exnuses    <--- [] ;;
ProcCall    stmt_p.exnuses    <--- match proclokup $stmt_p.procenvir $IDENTIFIER.lex with
                                   (_,_,_,exnuses) -> exnuses ;;

Try         stmt_p.exnuses    <---
   (* The exceptions which are caught should be removed from the set *)
   let myuses = list_diff $stmt.exnuses $catches.caughtexns in
   myuses @ $catches.exnuses ;;

Raise       stmt_p.exnuses    <---
   [($IDENTIFIER.lex, exnlokup $stmt_p.exnenvir $IDENTIFIER.lex)] ;;

Catches     catches_p.exnuses <--- $catch.exnuses @ $catches.exnuses ;;
Catches1    catches_p.exnuses <--- $catch.exnuses ;;
Catch       catch_p.exnuses   <--- $stmt.exnuses ;;
```

The `procexns` attribute

synthesised

type: `(string * Internalname.t) list`

visible on: exns

This attribute contains the list of exceptions that a procedure is declared to throw. This list is compared with the list of exceptions that are actually thrown. Any discrepancies are enumerated by the `exnerrors` attribute.

```
Exns     exns_p.procexns    <--- $exnlist.procexns ;;
NoExns   exns_p.procexns    <--- [] ;;
ExnList  exnlist_p.procexns <--- $exn.procexns :: $exnlist.procexns ;;
ExnList1 exnlist_p.procexns <--- [$exn.procexns] ;;
Exn      exn_p.procexns     <---
   ($IDENTIFIER.lex, exnlokup $exn_p.exnenvir $IDENTIFIER.lex) ;;
```

The `caughtexns` attribute

synthesised

type: `(string * Internalname.t) list`

visible on: catches

This attribute enumerates the exceptions that are caught by the “with” section of a “try-with” block.

```
Catches  catches_p.caughtexns <--- $catch.caughtexns :: $catches.caughtexns ;;
Catches1 catches_p.caughtexns <--- [$catch.caughtexns] ;;
Catch    catch_p.caughtexns   <---
   ($IDENTIFIER.lex, exnlokup $catch_p.exnenvir $IDENTIFIER.lex) ;;
```

The `code` attribute

We have defined two new statements, so we need to define their `code` attribute. The “try-with” statement translates to a `switch` statement that switches on the kind of exception raised. If no exception was raised then the code jumps past the switch statement. The translation of the “raise” statement first updates the value of the global variable `exception`. Then it jumps to the appropriate “catch”. It may be that the exception was raised in a sub-routine. in this case the “raise” statement simply returns from the sub-routine. Below we describe how the translation of procedure calls has been modified to collaborate with this behaviour.

```
Try      stmt_p.code    <---
   let endlabel = Internalname.mkstring (Internalname.create "l") in
   C.Block([$stmt.code
                 ;C.Goto(endlabel)
                 ;C.Label(Internalname.mkstring $catches.jumplabel)
                 ;C.Switch(C.VarUse("exception"),
                                 $catches.code)
                 ;C.Label(endlabel)
                 ]) ;;

Raise    stmt_p.code    <---
   let exn_name = exnlokup $stmt_p.exnenvir $IDENTIFIER.lex in
   C.Block([C.Assign(C.VarUse("exception"),
                     C.VarUse(Internalname.mkstring exn_name));
      try let label = InternalnameMap.find exn_name $stmt_p.exncontext in
          C.Goto(Internalname.mkstring label)
      with Not_found -> C.Return
    ]) ;;

Catches  catches_p.code <--- $catch.code :: $catches.code ;;
Catches1 catches_p.code <--- [$catch.code] ;;
Catch    catch_p.code   <---
   let exn_name = exnlokup $catch_p.exnenvir $IDENTIFIER.lex in
   let exnvar = C.VarUse(Internalname.mkstring exn_name) in
   let reset = C.Assign(C.VarUse("exception"), C.VarUse("NOEXCEPTION")) in
   C.Case(exnvar,[reset; $stmt.code; C.Break]) ;;
```

Modifications to previously defined attributes

Extending the language with exceptions has consequences for some of the existing attributes. Three attribute computations need to be redefined.

Firstly, the code attribute for procedure calls needs to be modified. After a call to a procedure that may raise an exception, we need to check if an exception was raised. The C translation does this by switching on the value of the global variable `exception`.

```
ProcCall stmt_p.code <---
   match proclokup $stmt_p.procenvir $IDENTIFIER.lex with
     (newname, _, extraparams, exnuses) ->
   let convert_extraparams = List.map (fun name -> C.VarUse(Internalname.mkstring name)) in
   let proccall = C.ProcCall(Internalname.mkstring newname,
                             $paramlist.code @ convert_extraparams extraparams) in
   match exnuses with
     [] -> proccall
   | _::_ -> let mkcase = fun (name, id) ->
             let jumpcmd = try let label = InternalnameMap.find id $stmt_p.exncontext in
                               C.Goto(Internalname.mkstring label)
                           with Not_found -> C.Return in
                 let exnvar = C.VarUse(Internalname.mkstring id) in
                 C.Case(exnvar, [jumpcmd]) in
             let exn_switch =
                 C.Switch(C.VarUse("exception"),
                          List.map mkcase exnuses @
```

```
                    [C.Case(C.VarUse("NOEXCEPTION"), [C.Break])]) in
            C.Block([proccall; exn_switch]) ;;
```

Secondly, the procedure environment needs to be modified. It needs to contain an extra piece of information: the list of exceptions that the procedure might raise.

```
ProcDef decl_p.procdefs <---
   let myuses = VarInfoSet.diff $stmt.varuses
                                (varuses_of_paramdefs $paramdeflist.paramdefs) in
   Stringmap.singleton $IDENTIFIER.lex
                       ($decl_p.internalname,
                        List.map (cp snd paraminfo_of_paramdef) $paramdeflist.paramdefs,
                        List.map fst (VarInfoSet.elements myuses),
                        $exns.procexns) ;;
```

Thirdly, at the top level the C translation needs to define an `enum` of all the exceptions.

```
Program program_p.code <---
   let exnalldefs = $stmt.exnalldefs in
   ( match exnalldefs with
       [] -> []
     | _::_ -> let mkenumentry exndef = (Internalname.mkstring exndef, None) in
               [C.Enum("exception_enum",
                       ("NOEXCEPTION", None) :: List.map mkenumentry exnalldefs)
               ; C.Globalvar(C.Typename("exception_enum"), [C.Varname("exception")])
               ]
   ) @
   $stmt.proccode @ [C.Function(C.Void, "main", [], [$stmt.code])] ;;
```

Bibliography

[1] Cantu, M. (October 1999) *Mastering Delphi 5*. Sybex.

[2] Cheatham, T. J. (1966) "The introduction of definitional facilities into higher level programming languages." In: *AFIPS (Fall Joint Computer Conference, 29)*. (pages 623–637).

[3] Dueck, G. D. P. and Cormack, G. V. (April 1990) "Modular attribute grammars." *The Computer Journal*, **33**(2): pages 164–172. See also: research report CS-88-19, University of Waterloo (May 1988).

[4] Engler, D. R. (October 15–17 1997) "Incorporating application semantics and control into compilation." In: *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*. Berkeley: USENIX Association, (pages 103–118).

[5] Farnum, C. (1992) "Pattern-based tree attribution." In: *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*. New York, NY, USA: ACM Press, (pages 211–222).

[6] Feldman, S. I. (August 1978) "Make – A program for maintaining computer programs." Technical Report Computing Science Technical Report No. 57, Bell Laboratories.

[7] Fraser, C. W. and Hanson, D. R. (1995) *A Retargetable C Compiler: Design and Implementation*. Redwood City, CA, USA: Benjamin/Cummings Pub. Co.

[8] Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. (1995) *Design Patterns*. Reading, MA: Addison Wesley.

[9] Ganzinger, H. and Giegerich, R. (June 1984) "Attribute coupled grammars." *ACM SIGPLAN Notices*, **19**(6): pages 157–170.

[10] Harvey, B. (1997) *Computer Science Logo Style*. MIT Press, second edition.

[11] Johnson, R. E. and Foote, B. (1988) "Designing reusable classes." *Journal of Object-Oriented Programming*, **1**(2): pages 22–35.

[12] Johnson, S. C. (1975) "YACC: Yet another compiler compiler." *Computing Science Technical Report*, **32**.

[13] Johnsson, T. (1987) "Attribute grammars as a functional programming paradigm." In: Kahn, G. (editor), *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, (pages 154–173).

[14] Jullig, R. K. and DeRemer, F. (June 1984) "Regular right-part attribute grammars." *ACM SIGPLAN Notices*, **19**(6): pages 171–178.

[15] Kastens, U. and Waite, W. M. (October 1994) "Modularity and reusability in attribute grammars." *Acta Informatica*, **31**(7): pages 601–627.

[16] Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language, Second Edition*. Englewood Cliffs (NJ), USA: Prentice-Hall.

[17] Knuth, D. E. (1968) "Semantics of context-free languages." *Mathematical Systems Theory*, **2**: pages 127–146.

[18] Krasner, G. E. and Pope, S. T. (August/September 1988) "A cookbook for using the model-view-controller user interface paradigm in smalltalk-80." *Journal of Object-Oriented Programming*, **1**(3): pages 26–49.

[19] Kruglinski, D. J.; Shepherd, G.; and Wingo, S. (1998) *Programming Microsoft Visual C++*. Microsoft Press, fifth edition.

[20] Leavenworth, B. (1966) "Syntax macros and extended translations." *Communications of the ACM*, **9**(11): pages 790–793.

[21] LEROY, X.; RÉMY, D.; VOUILLON, J.; and DOLIGEZ, D. (November 1999) *The Objective Caml system release 2.04*. Institut National de Recherche en Informatique et en Automatique. Available from URL http://caml.inria.fr/.

[22] DE MOOR, O.; PEYTON-JONES, S.; and VAN WYK, E. (1999) "Aspect oriented compilers." In: *First International Symposium on Generative and Component-Based Software Engineering*.

[23] OUSTERHOUT, J. (1994) *Tcl and the Tk Toolkit*. Addison-Wesley.

[24] PETZOLD, C. (1999) *Programming Windows*. Microsoft Press, fifth edition.

[25] SCHNEIDER, J.-G. and NIERSTRASZ, O. (1999) "Components, scripts and glue." In: Barroca, L.; Hall, J.; and Hall, P. (editors), *Software Architectures – Advances and Applications*. Springer, (pages 13–25).

[26] SIMONYI, C. (1996) "Intentional programming: Innovation in the legacy age." Presented at IFIP Working group 2.1. Available from URL http://www.research.microsoft.com/research/ip/.

[27] STICHNOTH, J. M. and GROSS, T. (October 15–17 1997) "Code composition as an implementation language for compilers." In: *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*. Berkeley: USENIX Association, (pages 119–132).

[28] SWIERSTRA, D. and VOGT, H. (June 1991) "Higher order attribute grammars." In: Alblas, H. and Melichar, B. (editors), *Proceedings of International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*. Springer Verlag, (pages 256–296).

[29] VAN WIJNGAARDEN, A.; MAILLOUX, B. J.; PECK, J. E. L.; and KOSTER, C. H. A. (1969) "Report on the algorithmic language ALGOL 68." *Numerische Mathematik*, **14**(2): pages 79–218.

[30] VAN WYK, E. (2000) "Domain specific meta languages." In: *ACM Symposium on Applied Computing*.

[31] Zimmerman, M. W. (editor) (1998) *Microsoft Visual Basic 6.0 Programmer's Guide*. Microsoft Press.