Abstract

A fundamental part of the first year of a computing degree is learning programming. In order to become good programmers, students need to be able to concentrate on understanding the problems that they are solving, techniques and algorithms. This can be impeded by the need to learn the syntax of a complex programming language. Teaching languages provide simple syntax, but students are often not happy learning a language which they do not see as having commercial applications.

This report details the design of a teaching language, Kenya, and a translation system for converting code written in the teaching language into Java code. This gives the progressing student a stepping stone to Java, whilst letting the beginner concentrate on the principles of good programming. The report also encompasses the design of a development environment for writing programs in this language.

Acknowledgements

I would like to thank my supervisor Susan Eisenbach for her continued guidance throughout this project.

I would like to thank Christopher Anderson for his advice on constructing grammars, compilers and type checkers.

I would like to thank Ashok Argent-Katwala and Geoffrey Howie for the pertinent (and some not so pertinent) discussions that we have had over the course of this project.

Introduction and Motivation

Java[1] is a popular programming language in the computer industry today. Students want to learn Java, but its syntax is quite complicated for teaching introductory programming courses. Current methods of teaching introductory programming do not often start with object-oriented design, but rather concentrate on procedural programming and flow control, moving on to abstract data types and classes at a later stage. It is not possible to write a program in Java without writing at least one class (with a `main` method).

A better language for teaching with is something like Turing[2] which has a much simpler syntax, and allows programmers to do simple things in simple ways.

The "Hello World" Program in Turing

```
put "Hello World!"
```

The "Hello World" Program in Java

```
public class HelloWorld
{
   public static void main( String args[] )
   {
      System.out.println( "Hello World!" );
   }
}
```

Things we have to explain to new programmers to write the Turing program:

1. What `put` does
2. That strings need to go in double quotes

Things we have to explain to new programmers to write the Java program:

1. What a class is
2. What `public` means
3. What `static` means
4. What `void` means
5. What a method is and how it takes parameters
6. What a library is

7. What the dot operator does
8. What `System.out.println()` does
9. What curly brackets are for
10. What the semi-colon is for
11. That strings need to go in double quotes

It can be seen from this example that teaching introductory programming concepts is much easier with a language like Turing, certainly in terms of getting people to the stage where they can write their first program.

However, Turing is not a useful language in the "real world". People want to learn languages that they can actually use to write large applications, and put on their CVs. It can be argued that once the basic principles of good programming have been instilled through an introductory programming course, it should be easy to learn a second language. However, students often complain about having to learn with "toy" languages and would rather be learning a "real" language.

The aim of this project is to design a language which is simple enough to use to teach introductory programming effectively, but which the development environment will *translate* into the Java code which would be written to achieve the same task. The syntax of the language will be kept simple but will borrow from Java where appropriate. This should allow a programmer to progress easily onto writing Java at a later stage, whilst allowing them to concentrate on the principles and structures behind their programs at earlier stages, without getting bogged down in the overheads presented by writing the program in full Java.

The Current Situation

Currently in the Department of Computing at Imperial College, introductory programming is taught using a language called Turing[2]. The Department would like to move to a more Java-like system, to make it easier for students to move on to programming in Java, which is more applicable to real world problems.

The results of a brief survey of introductory programming techniques at other universities are included in Appendix D.

The Turing System

The Turing language has a lot of features which make it a good choice for teaching programming. Unfortunately it is not a language in which commercial software is developed as it does not tend to be used outside of educational institutions. For this reason, students are often displeased by having to learn Turing as they do not believe that it will be useful in the long run and feel that they would be better off learning a language which is more popular in industry, such as C++ or Java.

If fact, students will be much better off learning the principles of programming using a simple language. It is far more important that students have a thorough understanding of how to solve problems and how to construct elegant and efficient algorithms in their programs than to know the syntax of a certain language. Once the basic principles of good program design and implementation have been learned, it is easy to take these principles and apply them in different languages.

Initially, students need to be able to focus on the way that they are solving problems and not on remembering complex syntax. This is why a less powerful language with a less complicated syntax, like Turing, makes a good tool for teaching and learning the principles of computer programming.

The Turing system that is currently used has an Integrated Development Environment (IDE) which can be used to edit code, to compile it (into a byte code which is then interpreted by a virtual machine), to execute a program and to step through the execution an instruction at a time. The ability to step through the execution, and also to examine the contents of variables during that execution, makes debugging programs much easier as it allows the programmer to observe the path of control through their programs.

A Java-like Teaching Language

The Turing system has many features which make it a good teaching language, as have been described above. There is no substitute for learning good techniques and thinking about structure and algorithms when it comes to learning programming. The use of a teaching language makes it easier for students to develop these key skills without complex syntax getting in the way. However, programmers will soon want to move on to a more commercial language. At the time of writing, the language which is seen as most desirable by employers, and on which most research and development of software engineering techniques is being based, is Java. The learning curve in moving from Turing to Java is quite steep, as few pieces of the syntax are shared, and some of the object-oriented concepts in Java are hard to grasp quickly. The severity of this learning curve could be reduced if a teaching language was developed which shared the syntax of Java (or at least followed its general style) for expressing common constructs. Better than this would be a system which would take code written in a teaching language, and automatically translate it into the Java code that a Java programmer would write to complete the same task. This would allow a novice Java programmer to see how constructs from the teaching language map to Java code. If this could be combined with a good Integrated Development Environment then this would be a very good tool for teaching introductory programming with a view to later moving on to Java. The design of such a language and development environment is the aim of this project.

Naming the Language

All programming languages need a name. The programming language being designed in this project can serve as an introduction to Java, something to try before hitting the hard stuff. Taking a coffee analogy, I have chosen the name *Kenya* for the language, as this is a slightly milder coffee than Java (see www.starbucks.com).

Previous Attempts to Solve this Problem

The fact that Java is not the easiest or best language for teaching programming has been recognised by a number of people, and there have already been a few attempts to create languages and environments to help with this.

The JMac System

A previous student of the Department of Computing at Imperial College worked on such a system for a project. The result of their work was a system called JMac[4]. The main feature of JMac is a very nice Integrated Development Environment created using the Java Swing classes. This IDE is very well designed and it would be hard to find any substantial improvements to make upon it in terms of usability.

JMac also provides a simple programming language. When a program has been written the system translates the code into Java, compiles and runs it. However, the translation between JMac and Java is not as clean as it perhaps could be. For instance, the following is the code for the "Hello World" program in JMac:

```
print("Hello World");
```

This is fine, and perfectly simple, but asking the system to parse this code and create Java produces the following code:

```
/**
 *  A JMac Program
 */

class JMacProgram extends JMacLibrary {

      ///////////// Variable Declaration ///////////////

      /////////////  Array Declaration   ///////////////

      /** Constructor */
      JMacProgram() {

            //////////////     Program Code  //////////////
            print("Hello World");
      }
```

```
    /////////////  Method Declaration /////////////

    /** The main entry */
    public static void main(String[] args) {

        /* Creates a new JMacProgram */
        JMacProgram myJMacProgram = new JMacProgram();
    }

    ///////////// Advanced Program Code /////////////

}
```

It is not at all obvious to an introductory programmer what this does. Consider this program in terms of the the kinds of concerns previously raised about the problems for an introductory programmer of understanding some of Java's complexities: what is a `class`? What does `extends` do? What is a constructor? ...

Even an experienced Java programmer cannot understand from this code alone how the program works, as no source code is shown for the JMacLibrary class and so it is not known what functionality this provides. Also, the JMac program is simply quoted verbatim in the Java program, so it cannot be seen how the same functionality is achieved in Java.

This is not very useful as a step towards learning Java. It would be much better if on parsing the JMac program some Java code was produced that was equivalent in function to the original JMac program and was self contained (i.e. it should be the Java code that a programmer would write if they were trying to solve the same problem in Java as they have just solved using the simpler language.)

BlueJ

BlueJ[5] is a research project, supported by Sun Microsystems, in the "Blue" group at the School of Network Computing, Monash University, Australia. The aim of BlueJ is to provide an easy-to-use teaching environment for the Java language that facilitates its teaching to first year students. The designers of BlueJ have identified several problems with existing environments and aim to overcome these.

- **A lack of object-orientation in the environment -** BlueJ is designed to teach Java, and from an object-oriented standpoint from the start. Consequently a prime consideration in its design was to create an environment which supports and promotes the development of object-oriented programs. This is different to the aim of the system to be designed in this project. Here we are starting with a procedural programming approach, rather than an object-oriented one.

- **The environment is under or overkill -** Considerable problems are caused by an environment that provides tools that are too complicated, too minimalistic, or just the wrong tools.
- **The environment focuses on user interfaces -** Many environments use graphics to allow the user to design graphical user interfaces. BlueJ's environment concentrates on using graphics to illustrate the class structure of a program, making it easier to understand.

BlueJ also overcomes two problems associated with teaching Java caused by the language itself, namely the need to write a *main function*, and the difficulties with textual input and output. This project will also address these issues.

JJ

JJ[6] is a programming language and environment designed for learning Java. It was developed at Caltech by David Epstein and John Motil. JJ is a subset of Java (although it has syntactic differences), and was designed to offer the ideal syntax for teaching programming to first year computer science students. The designers had four main goals:

- JJ should be an ideal language for beginners
- JJ should be an introduction to Java
- A language for beginners should be easy to read, close to natural languages and devoid of unnecessary punctuation
- JJ should not have inheritance

JJ is an introduction to Java, and shares with this project the technique of translation from the teaching language to Java. The translation is done line by line, one line of JJ translating to one line of Java. However, although JJ is supposed to be a subset of Java, most of the keywords, and some of the syntax, are substantially different. For instance, the designers decided to apply a "command word rule" to the language design. This rule states that every command begins with a reserved keyword. There are no semicolons, each command must be on a separate line. There are no curly brackets. Multiline constructs are defined using `If` and `EndIf`, `Class` and

`EndClass` `Routine` and `EndRoutine` etc. The way in which code is indented is also important, as corresponding If and EndIf pairs must be aligned to the same column. JJ's designers have expressed the opinion that students find the word "variable" confusing, and so have used the word "box" in its place. Variable declarations are a case where JJ's syntax is substantially different to Java. JJ uses:

`Box` *name* `ofType` *type*

where Java would have:

type name`;`

Although I agree with the technique of translating a simple language into a more complex one as an aid to teaching, and will pursue this route in developing a teaching tool in this project, I believe that the weaknesses in JJ are in the dissimilarities between the JJ language and Java. During the time spent working with a language, certain things become ingrained, for instance the use of a certain character (e.g. a semicolon) as a line separator, or the order of the terms in a variable declaration. If a smooth transition between languages is to be ensured then, in my opinion, it would be more sensible to keep a common syntax and style between the two where this will not introduce overly complex constructs. I also think that the need for the student to learn a whole new set of keywords when making the transition will present a barrier to moving easily to Java.

JJ enforces a certain style of indenting. The way that code is laid out is very personal to individual programmers (although perhaps more of an issue to the more experienced programmer) and being forced to use a certain layout may prevent them from thinking about the problem as clearly as they might if it were possible to lay out their code with whatever spacing they wished.

Compiler Compilers

Parser Generation

Computer programs written in high or medium level languages are most often represented as strings of characters in text files. It is not possible for software to process raw sets of characters without some knowledge of what various combinations of characters forming keywords, identifiers etc (known as "tokens"), mean in terms of the language in which the program has been written. In order to do anything useful with the contents of these files, the characters need to be analysed and built into some abstract representation of the program, describing its structure, which can be processed[7][8]. Often this abstract representation is translated into some differing textual representation, e.g. the conversion from C source code to binary object code, or from Java source code to Java byte code.

Parsing involves grouping the tokens of a source program into grammatical phrases. The parser takes as an input the textual representation of the source program and constructs from it an abstract hierarchical representation of the same program, usually in the form of a tree. There are two stages to this conversion. The first is lexical analysis, which takes a stream of input characters from the textual representation of the program and converts them into a stream of tokens. The second stage is syntax analysis, where a parse tree is built from the stream of tokens which represents the program. This tree can be processed by further stages either to perform semantic analysis, for instance to check the correctness of the program, or to generate code (either machine code for a specific architecture in the case of a compiler, or source code for another high level language in the case of a translator).

As the translation from text to an abstract program representation is a problem which has been encountered many times before, it is now well understood how to construct programs to parse text into meaningful words and symbols and form so called Abstract Syntax Trees (AST's) representing the program. In fact, this problem is so common that software has been developed which will automatically generate a program to parse a text file into an AST, given a description of the language. Such software is called a "parser generator", or less accurately, a "compiler compiler". The description of the language to be parsed is usually given in the form of a "grammar", which defines what constructs (or "productions") can be used in the language, and what tokens comprise these.

There are many different parser generators available, some of which are described in the following paragraphs.

Lex and Yacc

Lex and Yacc[9] are fairly old UNIX programs. They split the parsing problem into two parts, Lex is essentially a lexical analyser (or "lexer"). It splits the source file into tokens. Yacc (Yet Another Compiler Compiler) finds the hierarchical structure of the program. The Yacc user specifies the structure of his input, together with code to be invoked as each such structure is recognised.

ANTLR

ANTLR[10] (Another Tool for Language Recognition) was written by Terence Parr who previously wrote PCCTS[11], but where PCCTS generates parsers written in the C language, ANTLR produces parsers written in Java or C++.

SableCC

SableCC[12] is another parser generator which produces Java code. An advantage of SableCC is that the code which it generates uses the Visitor Pattern[13]. This means that a new operation to be carried out on the AST can be added, without changing the classes belonging to the AST, by simply writing a new class containing methods which can be applied to each node in the AST. An instance of this class is then passed to each node's `apply()` method.

The parser is generated from a grammar for the language, given as a text file. The grammar defines the tokens and productions used in the language which need to be recognised.

SableCC has been used in this project because its use of the Visitor pattern gives an elegant design to the software to process the abstract syntax tree, and because it produces Java code, which is platform independent.

Language Design

The main concerns in the design of the language are:

- It should have a syntax which is easy to understand, remember and write.
- It should not be necessary to define a `main` function in a program.
- It should not be necessary to create classes or objects to write simple programs.
- The syntax of Java should be used where it seems suitably simple, so that the learner can see how to write the same programs in Java when they want to learn a new language.
- Enough suitable keywords and structures should be included to allow the language to have sufficient functionality for it to be used to solve all of the problems which might be set as an introductory programming exercise.

Data Types

It makes sense to share the concrete data types of Java so that no type conversion has to be done at the translation or compilation stage. The basic types supplied by Java are:

- boolean
- char
- byte
- short
- int
- long
- float
- double

Are all of these needed? It is probably not necessary to have all of these when writing introductory programs. The difference between a `float` and a `double`, or a `short` and a `long` is unlikely to be something that an introductory programmer should need to understand.

It would be useful to have a string type. Strings in Java are not basic types but objects. This can probably be made transparent to the introductory programmer.

A more suitable set of basic types might be:

- boolean
- char
- int
- real
- string

The term `real` has been used instead of `double`. It is necessary to have a floating point data type, but over complicated to have more than one at different precisions. A `double` is more flexible than a `float`, so it makes sense to use a `double` if only one of the two types is going to be included. It would be confusing to use the word `float` to represent this, as it would get translated to a Java `double` rather than a Java `float`. However, it does not seem sensible to call it `double` if there are no other floating point precisions available. This will only lead to the question "What is a single?". Calling the type `real` seems a good solution. An introductory programmer is much more likely to be familiar with the concept of a *real number* than a *double precision floating point number*.

Should `string` have a capital letter? In the introductory language a string is presented as a basic type. None of the other basic types have capital letters. In Java, String is a class, not a basic type, and classes start with capital letters (`String`). If `string` is left uncapitalised then it may be confusing when moving to Java. If it is capitalised, it will be incongruous, and require a complex explanation as to why it is that way. At the current time, I am of the opinion that `string` should not be capitalised.

Variables and Constants

In Java variables are declared like this:

```
int a;
int b = 100;
```

This seems good and simple, and will be used in the introductory language. The use of the semi-colon as a statement separator will be considered later.

In Java constants are declared like this:

```
{ public } static final int a = 100;
```

This seems complicated and difficult to explain. In the introductory language a `const` keyword will be provided, which will map to `static final` in the translation to Java.

```
const int a = 100
```

Records and User Defined Types

Records (structures containing a set of other types) e.g. a Point containing an x and a y co-ordinate, both of a basic type, are useful. In Java this would be a class.

A large part of programming and software engineering is about finding a good abstraction model. Writing a program to fit this model is helped a lot by the provision for user defined Abstract Data Types. These are again implemented using classes in Java.

A point class in Java:

```
class Point
{
   int x;
   int y;
}
```

In Java, the contents of the record (object) are accessed in the following way, assuming p is a Point:

```
p.x = 4;
p.y = 3;
```

This method of defining a class (using a `class` keyword), and the use of the dot operator to access member variables, is fairly simple and widely used across a variety of languages, so it seems a good idea to keep these for the teaching language. The use of curly brackets to delimit the class definition seems an acceptable approach. Taking this approach here suggests using curly brackets for delimiting all blocks of statements (loops ...) as they are used in Java.

In C or C++ there is provision to provide a new name for a type. For instance the name "age" could be assigned as a synonym for "int". This is done using a `typedef`. Although this helps with modelling a problem, there is no provision for using typedefs in Java. It would be difficult to write code in Java which a `typedef` would map to, so unfortunately it will not be included in the introductory language at the moment.

Declaring Variables of User Defined Types

In Java, objects (instances of classes) are created using the `new` operator, in the following way.

```
Point p = new Point();
p.x = 2;
```

This creates an object on the heap, assigning memory dynamically at runtime. In C++ objects can be created on the heap in the same way, or they can be created on the stack in the following way:

```
Point p;
p.x = 2;
```

This uses the same syntax as declaring a variable of a basic type. As the introductory programmer should not need to know the difference between creating an object on the heap and creating it on the stack, it seems sensible to use the C++ stack creation syntax for the creation of variables of all types. The differences between basic and non basic types are thus made as transparent as possible to the programmer, who can use them in the same way. As objects cannot be created on the stack in Java, a declaration such as `Point p` would have to be converted to `Point p = new Point()` by the translator.

Arrays

Arrays are fairly fundamental data structures. They are usually referenced by an identifier and an index. For instance in C or Java, a number (or expression) is used in square brackets after the identifier, e.g.

```
numbers[4] = 12;
```

This seems a fairly straightforward notation. In Turing, a similar syntax is used, but with round brackets instead of square. To preserve consistency with Java, square brackets will be used.

In Java, arrays are objects, and need to be created using the `new` operator, e.g.

```
int numbers[] = new int[12];

String names[];
names = new String[12];
```

In the introductory language, arrays will be declared without this, in a C style:

```
int numbers[12];
```

and converted to the correct Java by the translator.

Statement Separators

To determine where one statement ends and another begins, some sort of delimiter is needed. It would be possible to use a newline for this, putting each statement on a different line. However, it is quite often desirable to change the layout of the program code to make it easier to read and show its structure more clearly. Using newlines to define new statements makes this difficult. Adding white space may cause a working program not to function. An alternative, as used in C, C++ and Java, is to use a semicolon (or some other special character) to separate statements. This allows arbitrary newlines to be inserted without affecting the function of the program, although it does mean that an extra character has to be added after each statement, and this does not look as tidy as it would without. The fact that Java uses a semicolon as its statement separator swings the decision to use a semicolon in the teaching language also. It is often the case that a lot of compiler errors in Java, and even more in C/C++, are caused by the programmer forgetting a semicolon at the end of a line. If the programmer gets into the habit of putting semicolons in right from the start then this may be reduced.

Conditionals

Conditionals are a fundamental part of any programming language. The most useful and generic construct is *if .. then .. else*. The Java syntax for this seems straightforward enough to include in a language for teaching, so the following syntax will be used:

```
if ( condition - a boolean expression )
{
   statements ...
}
else
{
   statements ...
}
```

Case

Case is a useful construct to prevent programmers having to write:

```
if ( a == 1 ) { statements }
if ( a == 2 ) { statements }
if ( a == 3 ) { statements }
if ( a == 4 ) { statements }
```

In Java the case construct is called `switch` and has the following syntax:

```
switch ( a )
{
  case 1 :
           text = "first case";
           break;
  case 2 :
           text = "second case";
           break;
  case default :
           text = "no other cases match";
           break;
}
```

If `a` does not match any of the cases, the default case is selected. The `break` statements are used to stop execution at the end of each case and jump to the closing curly bracket. If the `break` statement was not included at the end of case 1, and case 1 was matched, after case 1's statements had been executed, case 2's statements would also be executed, and so on until a break statement was reached. Some programmers find it an annoyance to have to include a `break` at the end of each of their cases, but it does allow for the possibility of leaving them out on purpose in order to let the execution drop through to the next case. In order not to prevent programmers from using this technique if they want to, it will be left to the programmer to put in the `break` statements rather than having the translator put them in automatically.

Loops

There are four common types of loops: *while* loops, *repeat* .. *until* loops, generalised loops and *for* loops. The first two are quite similar. With *while* the test for exiting the loop is done at the top (so the body of the loop may not be executed) and with *repeat* .. *until* the test is done at the end, so the body is always executed at least once. The Turing programming language provides a generalised loop in which the programmer can put the exit condition at the top or the bottom (or anywhere in the middle!) to determine the way in which the loop works. Most *repeat* .. *until* loops can be rewritten as *while* loops, so to aid consistency only a *while* will be provided.

```
while ( condition - boolean expression )
{
    statements
}
```

The other sort of loop to be considered is the *for* loop. This gives a number of iterations using an index variable. For example `for i from 1 to 10` or `for i from 10 to 1`. The syntax for this in Java is:

```
for ( i = 1 ; i <= 10 ; i++ )
{
    statements
}
```

This provides maximum flexibility from one construct without increasing complexity to do more sophisticated things (e.g. increment in different steps or have complex loop termination conditions).

Turing takes a slightly less sophisticated approach, using:

```
for i : 0 .. 9
    put i
end for
```

The Java approach provides much more flexibility, but the Turing syntax is far simpler. It is a difficult decision which of these is the more important consideration. I would argue for simplicity, consistent with all of the features of this new language, but at the same time it would be easy to make the language too restrictive, therefore not allowing more sophisticated programs to be written. While the language should be kept simple, it is also important that a large number of problems can be solved and techniques applied using it. My proposed syntax for the Kenya `for` loop is as follows:

```
for i = 0 to 9
{
    print i;
}
```

or

```
for decreasing i = 9 to 0 step 2
{
    print i;
}
```

The second case gives a loop counter which is decremented by 2 at each iteration of the loop.

It may be possible to allow both this format and the Java style format of the loop in order to let the programmer (or the teacher) choose which they feel is more useful or applicable in a certain case.

Procedures and Functions

Java calls procedures and functions "methods". Procedures are just functions which do not return a value (they return "void"). All methods are members of classes. Unless a method is declared `static`, it can only be called if an object of the class of which it is a member has been created. As at the current time the introductory language will not support object-oriented programming, all methods should be members of the class in which `main` is defined, and be declared `static` with package access, so that they can be called from any part of the program.

Input and Output

Textual output in Java, on the console at least, is most easily achieved by using the library functions `System.out.println()` and `System.out.print()` to print a line of text (with a newline at the end in the case of `println()`). To hide the library, the teaching language will provide functions `print` and `println()` which will translate to calls of `System.out.print()` and `System.out.println()` respectively in the Java.

Doing console input in Java is not simple. The cleanest way found to read say an integer from the keyboard into an integer variable is to do something like the following.

```
try {
     java.io.BufferedReader stdin =
        new java.io.BufferedReader(
        new java.io.InputStreamReader(System.in));

        String line = stdin.readLine();
        int i = Integer.parseInt(line);
    }
catch ( java.io.IOException e )  { System.out.println(e); }
catch ( NumberFormatException e ){ System.out.println(e); }
```

At the moment I think that the best way to deal with input is to provide functions with names like `readInt()` and `readString()`. If the user includes these in their program, a function wrapping code similar to the above will be included in the Java source code, and called at the relevant point. Another option would be to have a generic `read()` function which would translate to different Java functions depending on the type of the variable to which the result of the `read()` is being assigned. This is more complex to implement.

I/O will be interactive, but graphics will not be provided as these are not really a fundamental concept in programming, and tend to be highly platform and language dependent.

Operators

The following operators will be provided:

- =
- +
- -
- /
- *
- ^
- ==

- !=
- <
- >
- <=
- >=
- and
- or
- not

Generics

Generics[14] are a sophisticated concept. Java is an object-oriented language and (almost) everything is an object (i.e. it *extends* the class Object). When programs deal with large numbers of objects, they tend to hold them in various kinds of containers (like Vectors, HashMaps etc). Any sort of object can be put into a container and got out again later. However, if you put in say a Dog, where Dog is a class that you or someone else has defined, and try to get it out again, you get out an Object. This is because containers hold Objects. They do not remember the more explicit type of each Object put into the container, and so they can only give an Object back. It is up to the programmer to remember the type of the objects they put into the container, and convert them back to this type using a *cast*.

In Java this looks like:

```
Vector v = new Vector();
Dog d = new Dog();
v.add(d);

Dog e;
e = (Dog)v.elementAt(0);
```

The cast is the bracketed `(Dog)` after the assignment operator on the last line. This *coerces* the Object which comes out of the vector to a Dog, so that it can be assigned to `e`.

It is somewhat annoying for the programmer to have to remember the type of the objects in a container and cast them whenever they are extracted. A solution to this problem would be to have a class called DogVector which only contained Dogs. We could then be sure that any object extracted from a DogVector would be a Dog and therefore no cast would be necessary.

However, there will be other types of objects that programmers will want to store in vectors as well as Dogs (in fact anything that is an Object) and using this approach a different class would have to be written for each container for each type of object to be contained. Every time a programmer defined a new type they would have to define a new set of containers to put them in.

Generics offer a solution to this problem by providing the possibility of having containers that are parameterised by type. That is, we can say we want a Vector < A > . This means we want a Vector, but that everything it contains will be of type A (where A could be Dog, Date, String ...) . The parameterised container then deals with any type coercion necessary.

C++ offers generics in the form of *templates*. At the moment[15] Java does not have generics, but compilers are available which will compile a superset of Java, including parameterised types. GJ (Generic Java)[16] is such a compiler. It would not be difficult to produce GJ code as the translation from Kenya rather than Java (GJ is a superset of Java, so the code would be pure Java if generics were not used in the Kenya code). This would allow programmers to use the feature, removing much of the need for casting, one of the less elegant features of Java.

The use of parameterised types is quite an advanced concept, and it is questionable whether they should be included in a teaching language. However, I think that they should be included as the novice can choose not to use them. When they do come to work with containers, the concept of the parameterised type can be explained just as easily as the need to cast objects when they are extracted from containers.

Issues with the Generated Java

The software developed in this project will translate the Kenya code written by the programmer into Java so that it can be compiled and run. (In fact, in order to allow generics as detailed in the previous section, the code produced will actually be GJ, not Java, but as GJ is a superset of Java, I shall use the term Java to describe the code produced unless specifically talking about generics.)

Qualities of the Code

The main objective for the Java was that it should be as near as possible to the code that a Java programmer would write from scratch to solve a problem. Often generated code is very obviously generated. It is often more verbose and less elegant than code written by a human. I wanted to make sure that the code generated from Kenya is clear and simple, so that it is as easy as possible for the novice programmer to understand what the Java does, whilst providing a good example of how to write good Java code.

I also wanted to show exactly what code was necessary. A different approach would have been to provide a lot of library code which could be called, allowing the generated code to be kept small. However, I wanted to avoid providing "black box" functionality hiding the implementation from the user. The idea is to allow the programmer to see how things work so that they can do them for themselves.

Going against this is the fact that there is a Java paradigm of code re-use. Once you have written code to do something once, you do not write it again, but import it from the previous project. This points towards the use of library code. A compromise needed to be reached.

Library Routines

I have decided to provide the routines for getting input into a program (reading numbers and strings from the keyboard) in a library. Doing input in Java is not simple. The simplest code to read an integer that I can come up with is still quite complex (see the language design section).

The code could be wrapped in a function:

```
static int read()
{
  try {
       java.io.BufferedReader stdin = new
java.io.BufferedReader(new
java.io.InputStreamReader(System.in));
       String line = stdin.readLine();
       return Integer.parseInt(line);
  }

  catch (java.io.IOException e) { System.out.println(e); }
  catch (NumberFormatException e) { System.out.println(e); }

  return 0;
}
```

A different function is required for each type of data to be read. A Java programmer would probably write a wrapper class called something like IntReader with a static method read() or they would write a Reader class which contained several methods: readInt(), readReal() etc. They would then use this class in any program that they wrote in future, using an `import` statement to include the class, or just by putting a copy of it in the same package.

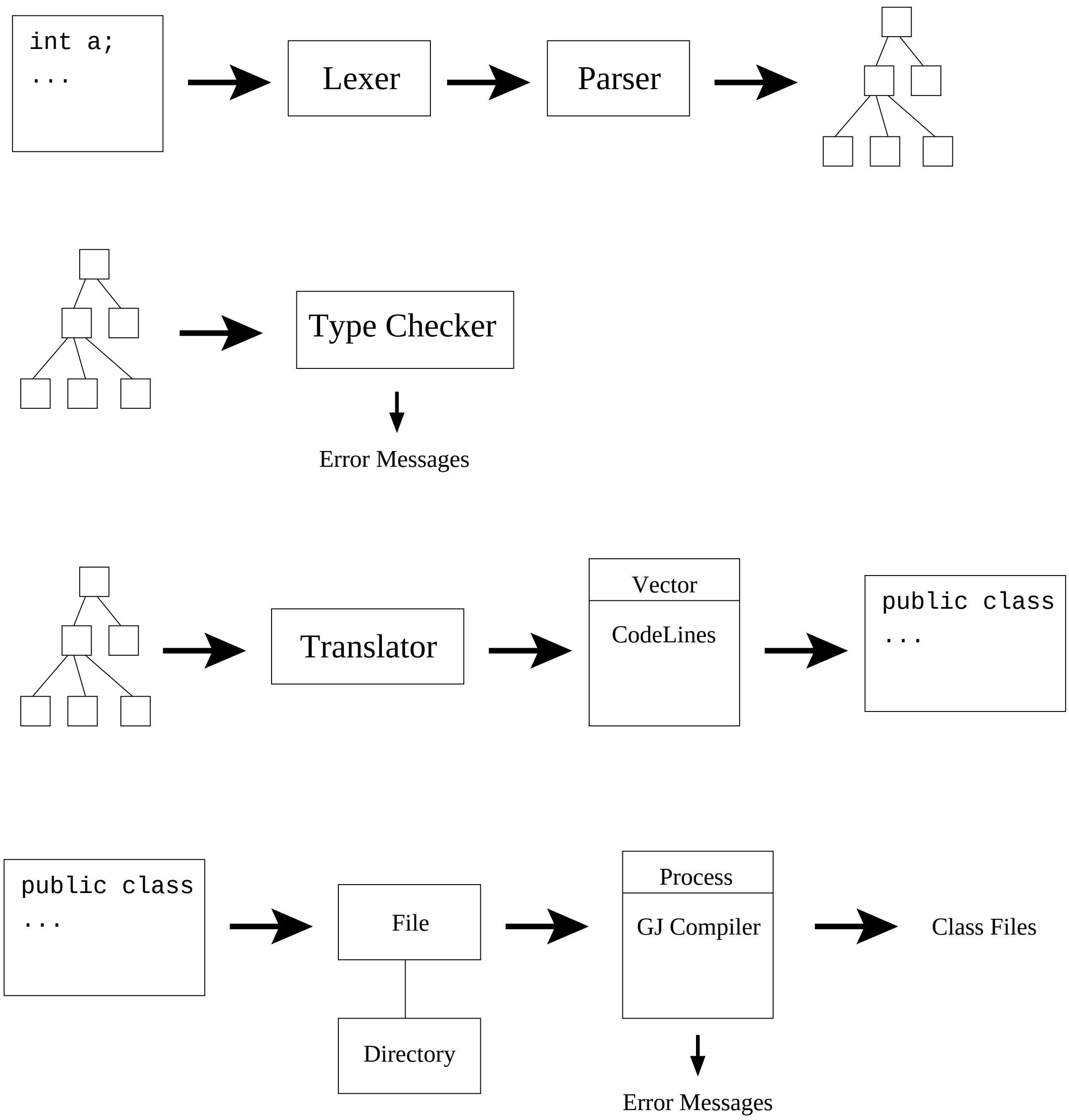
I have decided to write separate classes IntReader, DoubleReader and StrReader which are in a package called kenya.io . The use of the readInt(), readReal() or readStr() functions in a Kenya program causes the relevant class (or the whole package) to be imported into the generated Java program.

Software Design

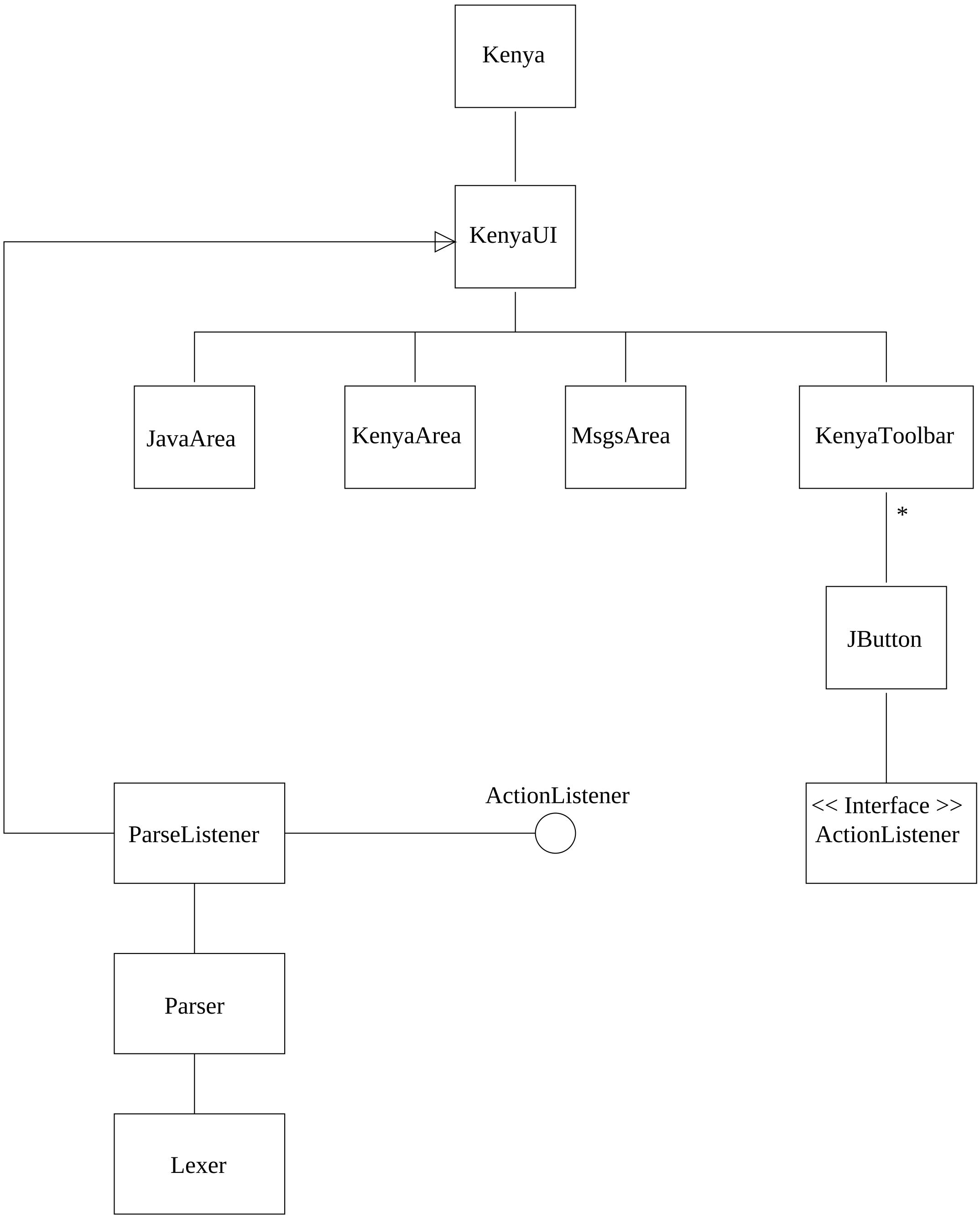
The software designed in this project takes a Kenya program and processes it in several ways. It builds an abstract representation of the program which can be checked for correctness and translated to Java code to be compiled and run. The details of the different stages of processing are given in the following sections. This section describes the overall structure of the system and how the different parts fit together.

The flow of information through the system is shown in the figure on the next page. A textual representation of the Kenya source code is typed in by the user. This is transformed into an abstract syntax tree by a lexical analyser and a parser. The tree is processed by a type checker to ensure correctness of the program, and by a translator to produce Java source code. The Java source code is then saved in a file in a directory and the compiler is run to produce a class file.

In order to provide platform independence, the software developed in this project has been written in Java. The diagrams on the following pages show some of the more important classes and how they are linked together. All of the different functional parts of the system are linked together by the user interface. The rationale behind the design of the user interface is described in another section. The user can enter code into the text areas and invoke other functions by pressing the buttons on the toolbar. Each JButton has an ActionListener which listens for mouse clicks and then calls appropriate routines to parse the code, compile it, open a file etc.

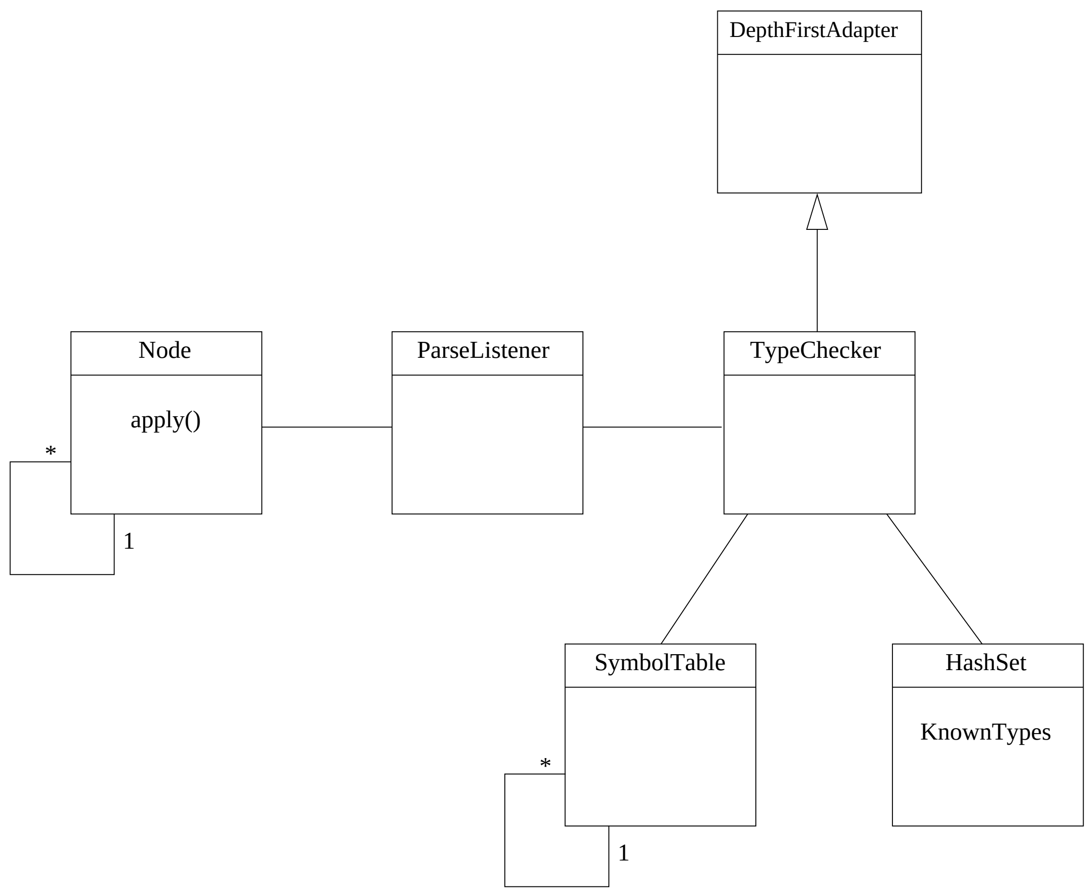


Information flow through the Kenya system



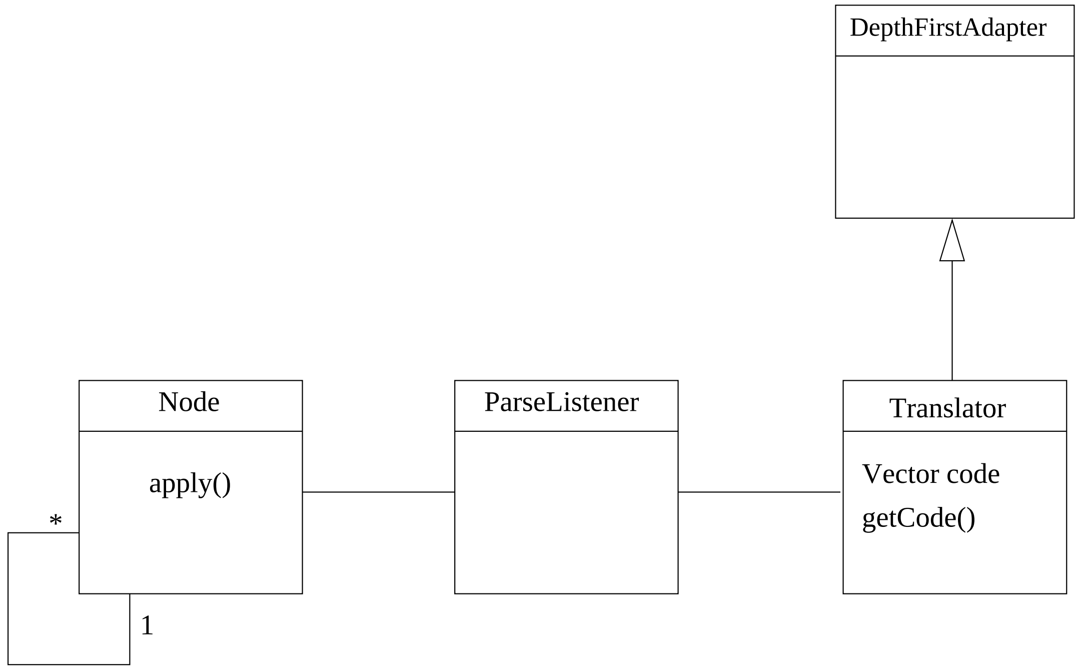
The overall structure of the user interface

When the ParseListener detects a mouse click on the "parse" button it creates a parser and a lexical analyser to construct an abstract syntax tree from the text in the KenyaArea. The tree is formed from instances of subclasses of the class Node. A TypeChecker object is then created and applied to the tree. The type checker maintains a set of known types and a symbol table mapping names to types.



The classes related to the type checker

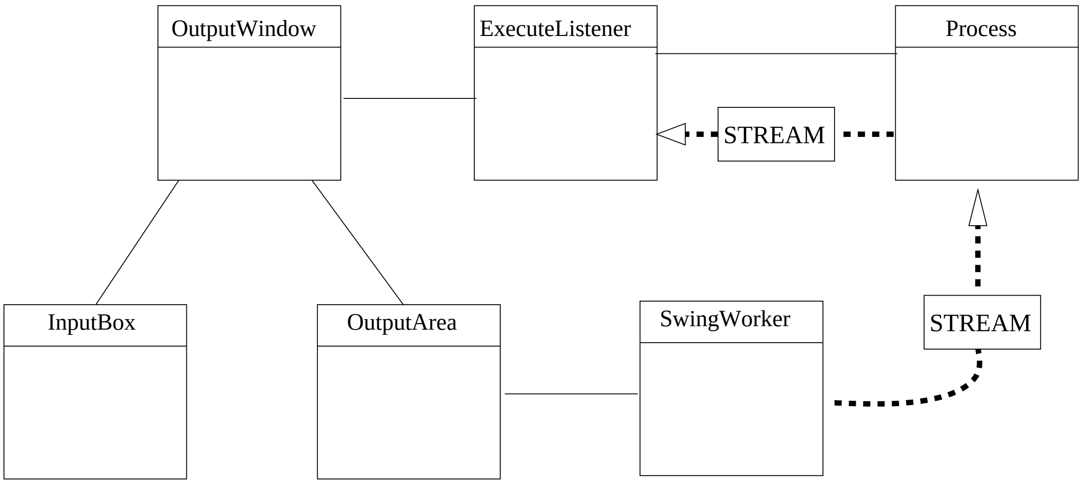
After the correctness of the program has been checked, a Translator object is created and used to process the tree to produce Java code. The translator adds each line of code that it generates to a Vector. Later the ParseListener can access the code using the translator's getCode() method. It writes each line of Java code into the JavaArea.



The classes related to the translator

Compiling the code causes the system create a directory and write the contents of the JavaArea into a file. The compiler is then invoked in an external process and produces class files.

When executing a compiled program, the system again runs it in a separate process but connects to the standard input and output streams for this process. This is so that output from the program can be piped into a window inside the Kenya user interface, and input from the user can be fed back to the process to allow interactive programs to be written and tested.



The classes related to executing a compiled program

Translation

A large part of the work in developing the Kenya system was developing a way to translate Kenya code into Java code. Although this is a mechanical translation, I wanted the Java code that was produced to look as "human generated" as possible. It should be as close as possible to the code that a human programmer would write to solve the problem in Java.

There should perhaps be a caveat here. Java is an object-oriented language, and the way in which "good" (or certainly large) programs are written in Java normally makes much use of this. Kenya is not an object-oriented language. There is provision for defining classes and creating instances of these classes, but there is no inheritance, and therefore no type polymorphism. (As it happens, this is the same as Visual Basic, which Microsoft claims *is* object-oriented, but I would argue against this). It is suggested that novice programmers should not be introduced immediately to the object-oriented programming paradigm, but that they should begin with a more traditional, structured approach. Because of this, the Kenya system will generate the Java code that a programmer would write to solve a problem if they were approaching it in a structured way, rather than an object-oriented way. There is not a straightforward mapping between a structured program and an object-oriented program to solve the same problem, as they are two different ways of thinking about solving the problem. This makes the task of writing a mechanical translator from structured code to object-oriented code extremely difficult. It also means that, were such a translation achievable, a novice programmer looking at the generated code would not easily be able to see, based on the way that they had been thinking about the problem, how the Java code worked.

To carry out the translation, first the text representing the Kenya program must be processed to form an abstract representation. Then the abstract representation can be processed, generating the relevant Java code from each construct and expression in the program.

Parsing

The parser for Kenya was generated using SableCC[12], as it outputs Java code which is easy to incorporate with the rest of the code for the project. Also, the use of the Visitor pattern allows for easy extension of the functionality defined over the AST without having to modify any of the generated code by hand, and also leads to an elegant solution.

Writing the Grammar

A grammar for SableCC is written using Extended Bachus Naur Form (EBNF)[17]. The grammar has the following sections (examples are given of the sorts of definition which might appear in each section):

```
Helpers
  digit = ['0'..'9'];

Tokens
  minus = '-';
  plus = '+';
  blank = ' ';
  number = digit+;

Ignored Tokens
  blank;

Productions

  expression =
    {plus}   expression plus number  |
    {minus}  expression minus number |
    {number} number;
```

Each of the constructs with an equals sign in the middle defines the name on the left hand side to represent whatever is on the right hand side. In the case of `digit`, this is the set of digits 0 to 9. The tokens give a name to each of the characters which may appear in the program source. Operators such as + can be used, as in the case of `number`. The use of + says that a `number` is a sequence of one or more digits.

Productions represent the way in which tokens can be combined to write programs. They can be defined recursively as shown in the example. There are several alternatives for what an expression may be. It may either be a number or an expression to which a number is added or subtracted.

Once a grammar has been written, SableCC will try to generate a lexical analyser and a parser for the language. Quite often during the writing of a grammar SableCC will report that it cannot generate a parser as it cannot resolve precisely which production a set of tokens represents. This is a *shift/reduce* or *reduce/reduce* error and is caused by an ambiguity in the grammar.

SableCC produces a so-called "bottom up parser". This means that the parser continues from left to right along a stream of tokens, pushing the tokens on to a stack. At each stage it tries to determine whether the elements at the top of the stack represent an entire production, or whether more tokens need to be examined in order to make an unequivocal decision as to what the string of tokens represents. If the top elements of the stack do represent a production, those elements are "reduced" exchanging the set of tokens for a symbol representing the production as a whole. If there is not yet enough information to make a decision then the next token in the stream is "shifted" on to the top of the stack and the process continues.

If at any point the parser is in a situation where it has multiple options and cannot make a decision based only on the contents of the stack and the next token, then there is a "shift/reduce conflict" (if the parser is unsure whether to shift or reduce) or a "reduce/reduce conflict" (if there are several possible reductions which could be made). See the Dragon book[8] for more details on bottom up parsing techniques.

When SableCC generates a parser, it checks whether either of these conflicts can arise. If they can then it will not complete the generation of the parser. The productions in the grammar must be rewritten until the parser will no longer be able to enter such a conflict state. Such conflicts are generally very frustrating and not straightforward to remove or prevent. A great deal of time was spent during this project working on the grammar to remove conflicts.

Analysis

SableCC takes a grammar definition and produces from it a Node class and a set of subclasses of it which represent each of the operators and constructs which can appear at each of the nodes in the syntax tree. It produces a lexical analyser and a parser which together process the textual representation of the source code to produce this tree. Everything which is a Node has a method with the signature `public void apply( Object o )`. This allows visitor objects to be passed to each node to perform a particular type of analysis or processing on them.

SableCC generates a class called DepthFirstAdapter which will perform a depth first traversal of the parse tree, without performing any operations on the nodes as it goes. This class can be extended and its methods overridden to process the tree.

The methods in DepthFirstAdapter have a signature of the following form:

```
public void caseAVariableDeclaration(AVariableDeclaration node)
{
    // do something
}
```

AVariableDeclaration is a subclass of Node. There is a caseXY method corresponding to each type of node which may be encountered during the traversal of the tree. The relevant case method is called depending on the type of the Node to which the DepthFirstAdapter is passed.

Most Nodes in the tree will have children. What children they have depends on the grammar. For example, the grammar might contain the following definition:

```
variable_declaration =  type identifier initialiser? semicolon;
```

Here we can expect a node of type AVariableDeclaration to have children representing exactly one type, one identifier, zero or one initialisers and a semicolon. This might be processed in the following way:

```
public void caseAVariableDeclaration(AVariableDeclaration node)
{
    addToSymbolTable( node.getIdentifier() , node.getType() );

    if  ( node.getInitialiser() != null ) {
         node.getInitialiser().apply( this );
    }

  // control returns here after the tree representing
  // the initaliser has been processed
}
```

Here we see that get methods are provided for all the possible children of the node. In the case of the initialiser, as it is optional, a test is done to see if there is one. If there is, the visitor is passed on to process the tree representing the initialiser (an initaliser will probably consist of at least an assignment operator and an expression, which may in itself be a complicated set of nodes) by calling the node's `apply()` method and passing it the DepthFirstAdapter. (i.e. `this`.) Programmers unfamiliar with the Visitor pattern may find it slightly counterintuitive that the `this` reference refers to the DepthFirstAdapter rather than the Node that is being processed. After the tree representing the initialiser has been processed, the flow of control returns to this node, as indicated by the comment in the code fragment above, showing that the tree is processed recursively.

Code Generation

Once the abstract syntax tree has been generated, it can be processed by the code generator to produce equivalent code in a different language, in this case Java (with GJ extensions, see the language design section).

The main difference between a Kenya program and a Java program is that a Kenya program comprises a list of statements, whereas a Java program consists of a set of class definitions, and has the following structure:

- A program consists of a set of class definitions
- A class definition consists of a list of attribute declarations and a list of method definitions
- A method definition will contain a list of statements

One of the methods defined must be called `main`. This is the point of entry into the program. In Kenya, the point of entry is simply the top of the program.

The main issues of concern when generating Java code from a Kenya program are therefore:

- To create at least one class
- To insert the code for the main program operation into the `main` method of this class

Other pertinent issues are:

- To create methods relating to any functions defined in the Kenya program
- To create Java classes relating to any classes defined in the Kenya program
- To add any necessary `import` statements to allow access to code necessary for the correct compilation of the Java
- To generate a correct translation of all statements

Style

One of the most difficult things to decide was how to write Java code in the style of a human. Different people have different coding styles, and there is no one "best" style. Style is concerned both the way that constructs are defined and combined, and with the layout of keywords on the screen (i.e. what the code looks like).

In a procedural language, a constant would be declared just like any other variable (possibly at the top of the file). A Java programmer is more likely to define their constants as being `static final` members of the class which they are writing. If a direct, line by line translation was made from Kenya to Java then constants would be defined in the body of the main method. The Java code would still compile, but it would not be a good example of how to write Java programs. To avoid this, in the Kenya implementation all of the constant declarations are pulled out of a Kenya program and defined as attributes of the class.

In terms of layout, a consistent layout was used, always starting blocks with the curly bracket on a new line after an `if` or a `class` keyword. In terms of the way in which the program itself is structured, this is largely going to be due to the way in which the user has solved their problem. However, there are a few Java idioms which can be applied when generating the Java code.

The placement of different elements within a Java program is also a matter of style. In the generated code the following pattern is followed:

```
class Classname {

  constants

  main method

  other methods
}

class Record
{
   attributes
}
```

Kenya classes are just records and so cannot have methods included in them, hence the class Record above has only attributes.

Having an order for generating the code for different elements which is not necessarily the same as the order in which they are defined in the Kenya program means that the nodes of the abstract syntax tree cannot simply be processed in order. Account must be taken of what the nodes represent and the tree must be rearranged to meet the required structure.

Implementing the Translator

The translator was implemented by writing a visitor which could be passed to the `apply()` method of each node of the syntax tree produced by the parser. The class Translator extends the class DepthFirstAdapter created by SableCC. The DepthFirstAdapter performs a depth first traversal of the tree, doing nothing to the nodes as it walks over them. By extending this class and overriding its methods, a visitor was created which traverses the tree and generates the Java code relevant to each node.

Initially the approach taken was to simply generate the code

```
public class Program
{
    public static void main ( String[] args )
    {
```

and then traverse the tree, generating the code for each statement and appending it to the program. When the traversal was complete, the program was completed by closing the method and the class

```
    }
}
```

I decided to call the main class Program in all cases. Class names are generally decided by the purpose of the code being written. It is not possible for a machine to infer from the Kenya code what the intent of the program is and summarise it with a suitable name, so I decided to stick with Program.

Function Definitions

For simple programs the above approach works well. However, when a function is defined in the Kenya program, its definition cannot simply be written out in the flow of the Java code. Doing this would put one method definition nested inside another. Java does not allow this as all methods have to be members of a class, even if they are static and can therefore be called without creating an instance of the class. The code which needs to be generated is of the following form:

```
public class Program
{
    public static void main ( String[] args )
    {
        statements
    }

    static void doSomething()
    {
        statements
    }
}
```

To achieve this, whenever a function definition is encountered, instead of continuing down that branch of the tree and generating the code for the method definition, the branch is copied into a list of function definitions, and the code generator moves on to deal with the next branch of the tree. At the end, the `main` is closed, but before the class definition is closed, the translator object processes each function definition which has been accumulated in the list individually.

All the methods for which code is generated are declared to be `static`, as an object of type Program will never be created. However, only `main` is declared as being `public`, as it is the only method which will be called from outside the package (by the Java Virtual Machine when the program is invoked). All other methods will be called from inside the same class, or the same package, and so can (and should) be left "friendly", without an access modifier.

Class Definititions

A similar issue to that raised by function definitions is raised by class definitions. In Java it is permitted to nest one class definition inside another, producing a so-called "inner class". However, it seems clearer (to me at least) to have all classes defined at the same level. There are situations in which inner classes are useful. These concern *callbacks* and cases in which *multiple implementation inheritance* is required (see ch 8 of Eckel[18] for a discussion of these) but these issues are far more complicated than anything that should concern an introductory programmer and I feel that the use of inner classes should be reserved for these more sophisticated uses, as they are not required in solving more everyday problems. To achieve this, class declarations are added to a seperate list, and the code generator applied to them at the end, as is the case with function definitions.

Java programmers often define each class in a separate file. The compiler insists that all *public* classes are defined in their own file, and that the file is named after the class. Programmers often take this to mean that *all* classes should be put in their own file. This is also caused by programmers making any class that is not private, public. The public modifier is only necessary when a class needs to be accessed by a class in a different package. Any classes that are only accessed by classes in the same package (the majority of cases) can be defined as "friendly", with no access modifer, and can be defined in the same file. It is my belief that, particularly for simple classes, it makes much more sense to include several classes in a file. This saves having to constantly switch between files or have multiple editor windows open, and allows the whole program to be seen at once. I believe this is valuable when trying to learn how a program works.

Class Names

As mentioned above, each translation of a Kenya program to Java produces a public class called Program. In order for the Java compiler to process this class, it must be saved in a file called Program.java. This only becomes a problem when the user gets to the stage where they want to work with the Java. They will end up with multiple files all called Program.java. If they change the name then the file will not compile, but if they do not change the name then they cannot keep the files in the same directory.

The user could change the name of the class to something other than Program and change the name of the file accordingly, but even if they do this there may be other problems. When a Java program is compiled, a .class file is produced for each class that has been defined. If two Kenya programs are written which each define a class Dog, these will translate to two Java programs which each define a class Dog. The user can change the name of the main class from Program to something else so that they can keep the two programs in the same directory for convenience. However, when they compile one of the programs using the Java compiler, this will create a file called Dog.class which will overwrite any Dog.class files produced by previous compilation of the other program. This will cause the first program to execute incorrectly unless the definitions of the Dog class in both programs were the same.

These difficulties can be overcome by using Java's package system. By defining a package corresponding to each Kenya program that is written, all the .java and .class files generated from that Kenya program can be put in the same package. In the filesystem this is implemented by creating a directory, so it does not allow multiple Program.java's in the same directory, but it does provide a neat organisation, and certainly avoids the conflicts indicated by the Dog example above. The best way to get a sensible package name before creating the directory is to ask the user to provide one. A `package` statement needs to be added to the top of the Java program to indicate which package the class is in.

Input

Kenya provides the facility for obtaining input in programs by providing the functions `readInt()`, `readReal()` and `readString()`. These translate to somewhat complex functions in Java, as can be seen in the language design section. Although in general in designing the Kenya system it has been my principle to present the user with all the code necessary to achieve the same functionality in Java, in this case I have decided to place the functions into library classes and import them into the namespace. This does not go against the principle of generating the code that a human Java programmer would write, as a human programmer would not rewrite classes that they had written before, but would reuse the code, importing it from another package. The code for the input routines will be supplied in the language reference for those who wish to know how they were written.

As the translator traverses the syntax tree, every time it comes across a call to an input function it adds an entry to a set. If only one type of input has been used, (e.g. just `readInt()`), that class is imported explicitly by placing the following command at the top of the program.

```
import kenya.io.intReader;
```

If more than one type of input has been used then the whole of the kenya.io package is imported into the namespace instead using the command

```
import kenya.io.*;
```

Type Checking

If a purely mechanical translation from Kenya to Java is carried out, and the resulting Java code presented to the Java compiler, any mistakes in the Kenya code will be echoed or amplified in the Java code and cause compilation errors. The errors reported by the Java compiler will be in relation to the Java code only and so will be useless, and probably confusing, to the novice programmer who is working with Kenya. They may not be interested in the Java stage at all. Errors during compilation will therefore signify nothing more useful than "There is a bug in your program." To aid this situation, the system should make checks for the correctness of the Kenya program before carrying out the translation. Syntactically incorrect Kenya programs (i.e. programs that do not conform to the grammar) will generate errors in the lexical analysis and parsing stages. Error messages are returned to the user at this stage, so incorrect code is not generated.

However, there may be still be problems with a program which is syntactically correct. The Dragon book[8] says *"A compiler must check that the source program follows both the syntactic and semantic conventions of the source language. This checking, called static checking (to distinguish it from dynamic checking during execution of the target program), ensures that certain kinds of programming errors will be detected and reported."* In the Kenya system two different static checks are considered: uniqueness checks and type checks.

Uniqueness Checks

Variables, constants, functions and types must have unique names in order that the compiler can tell which the programmer is referring to. For instance, the following Kenya code, although syntactically correct, should not be considered to be a valid program.

```
int a;
a = 4;
int a;
a = 6;
print a;
```

Here two variables with the name "a" have been declared. This error should be reported. Generating Java from this code blindly, without checking for the uniqueness of identifiers, would produce Java code that will not compile properly.

It is possible to have variables with the same name in different *scopes*. A local variable inside a function may have the same name as a variable in the main body of the program. Any use of the variable name inside the function body will refer to the local version, leaving the previously declared version unaltered. Once the program's execution has left the scope of the function, references to that variable name will refer to the global version again.

It is also possible to give a field of a class the same name as a global variable. When referring to the field the programmer will add a qualifier which will distinguish it from the global variable, as in the following example:

```
class Dog
{
   int age;
}

Dog rover;
int age;

age = 22;
rover.age = 6;
```

The possibility of having the same name refer to different variables in these situations must be taken into account when performing uniqueness checks.

Type Checks

Type checking is the process of analysing a program to ensure that the types of expressions are consistent. For instance if a variable is declared as being of type int then it should not be assigned a real value (or a string or any other type). To perform these checks the type checker needs to keep a record of the type associated with each name and check this type each time the name is referenced in the program.

Another check which should be made when a variable is declared is that it is assigned a type which exists. If the programmer declares a variable of a user defined type, then they must also have defined the type. This check will also pick up simple typographical errors in type names.

Type Checking and Type Inference

There are two possible approaches which can be taken to verify that the types in a program are correct. One is known as *type checking* and the other *type inference*. These are best illustrated in terms of an example. Say we are checking the following code fragment:

```
int a;
a = 4 + 5.7;
```

Let us firstly take a type checking approach. Here, the variable `a` has been declared as having type `int`. We consequently examine the assignment statement. We know that `a` has type `int` so we check that the assignment which is being assigned to `a` also has type `int`. For an addition to have type `int` both of the addends must also be of type `int`. First we check 4, which is an integer as we expect, but then we check 5.7, which is a real number, and therefore we have a type error.

A type inference approach works the other way around. We expand the expression as far as possible and find the types of the end nodes, in this case 4 and 5.7, `int` and `real` respectively. These two are added, and so we infer that the result must have type `real`. This real value is assigned to a variable, and so this variable should also have type `real`, but we find that it does not, and so again we have a type error.

I have chosen to take a type checking approach in implementing the Kenya type checker.

Implementation of the Kenya Type Checker

The parser creates an abstract syntax tree representing the program as described in a previous section. The code which SableCC produces promotes the use of the Visitor pattern. This means that a TypeChecker object can be passed to the nodes of the abstract syntax tree in order to process it without writing any code inside the classes from which the tree is built. This offers the advantage of being able to alter the implementation of the type checker independently of the tree. If the grammar of the language is changed and the parser and AST classes regenerated, then only code in the TypeChecker class needs to be altered to take account of this change, rather than having to change code distributed throughout a number of different classes.

Each subclass of Node generated by SableCC has a method `apply()`. This method is void, (i.e. it does not return a value). Initially this seems to present a problem when trying to write a type checker as an obvious approach would be to process the nodes of the tree recursively by calling a method which returned a type. This is especially true if a type inference approach is taken. However, it is possible to overcome this when using a type checking approach by including an attribute in the TypeChecker class in which the expected type is stored. The type checker object is passed to the node to be checked, its type is compared to that which is expected, and another boolean attribute is set, depending on whether or not the types match.

A record needs to be kept of the type associated with each name referenced in the program. This is done by constructing a symbol table, and adding an entry to this table each time that a variable declaration is encountered while traversing the tree. The symbol table is implemented using a hashtable, and each entry is indexed by the name. The symbol table contains information about the type associated with each name, and also whether it was declared as a variable or a constant. It should be ensured that assignments are not made to constant values. Uniqueness checks are performed by checking that the key does not already appear in the hashtable before adding a new entry.

When a variable of a type which is a class is defined, as well as noting the type of this variable in the symbol table, the types of all of the attributes inside the class must also be recorded. This could be done by including the fully qualified names of all accessible attributes in the symbol table, but this is an inefficient implementation. It should not be necessary to repeat the information about the types of attributes in the table every time a variable of a user defined type is declared, as each of these will have the same set of attributes with the same types, even if they are assigned different values.

To avoid this, the approach which I have taken is, whenever a new class is defined, to create a separate symbol table for the scope of that class, and whenever a variable of that type is declared, to insert a reference to that local symbol table into the global symbol table. When a lookup is required, the qualified name can be split where the dots appear (as a qualified name will be of the form `varname.attribute`), the left hand side used to index the global symbol table, and the right hand side used to index the class specific one to find the attribute information. This indirection can be applied recursively so that classes may be nested to any depth desired by the programmer.

A set of known types is also maintained, so that whenever a declaration is encountered a check can be made to ensure that the type to be associated with the name is valid.

The User Interface

Although some programmers prefer to edit their code using a text editor like *emacs* or *vi* and run their compiler from the command line, a novice programmer is likely to prefer a more "friendly" interface. To this end, an Integrated Development Environment should be provided for Kenya.

Integrated Development Environments

There are many IDEs available for developing programs in languages such as Java, C, C++, Visual Basic ... They all have many different features, but tend to share a few common key elements:

- a code editor
- the ability to run the compiler from inside the IDE
- the ability to test and debug the execution of the program

Environments like Microsoft's Visual Studio[19] and Sun's Forte[21] offer many other features like class wizards and graphical resource or user interface editors. These more advanced features are likely to be confusing for the novice programmer. It would probably be better if an IDE targeted particularly at new programmers had only the essential features that are necessary to write, compile, debug and run a program without any other, potentially confusing, bells and whistles.

The JMac IDE

When the JMac[4] system, which was mentioned in a previous section, was developed, the main goal of the project was to produce a good IDE. It was originally my intention to use this IDE for Kenya, simply plugging in the code relevant to parsing and translating the Kenya language where previously the code for the JMac language was. However, after spending some time analysing the code for the JMac IDE. I felt that it would probably be easier to write my own graphical interface from scratch, as it was difficult to work out exactly how the JMac IDE code worked. This was not because the code was badly written. The large number of different classes and the fact that the Graphical User Interfaces is event driven, rather than following a step by step procedural execution, (so there is no real start or end to the program) make it hard to analyse. Also, the JMac IDE had some features which I thought were unnecessary, such as a class wizard. By designing my own IDE I could provide exactly the features that I wanted.

The Kenya IDE

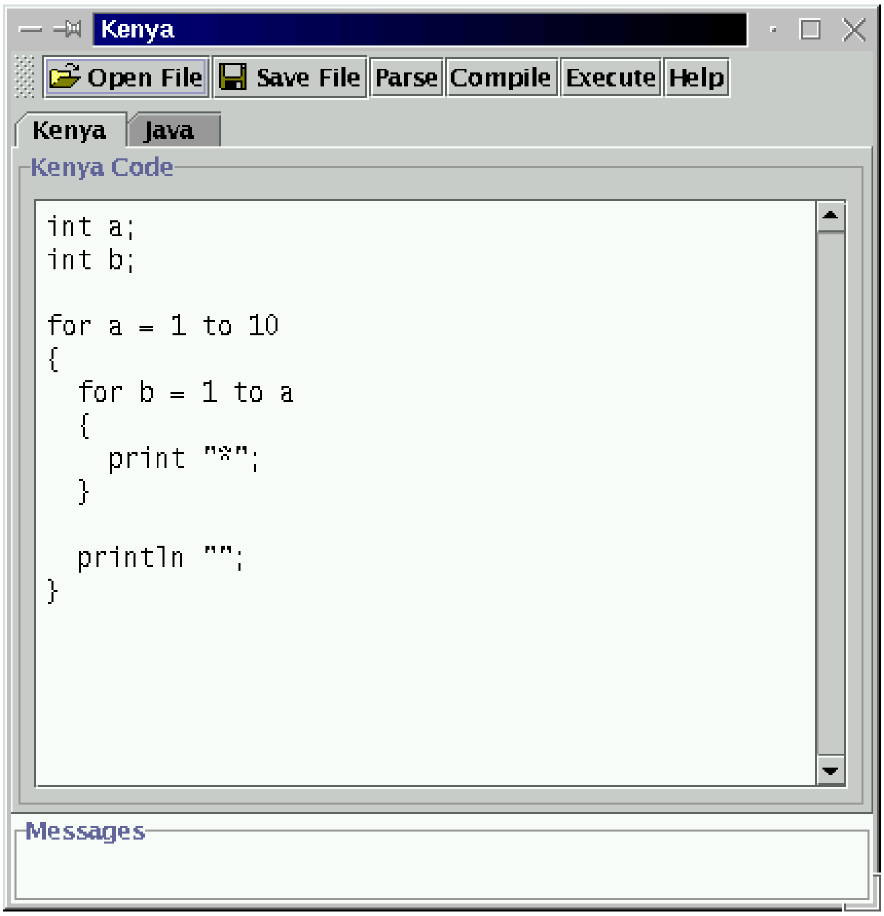
I wanted to keep the IDE for Kenya as simple as possible. The main features that are required are:

- areas in which Kenya and Java code can be edited
- a mechanism for accessing features (either a menu or a toolbar)
- some sort of display of status information
- a window in which to view the output of a running program
- a system to provide help to the user

A lot of existing IDE's (and other types of software) contain a large number of features. The way in which people learn new pieces of software tends not to be by reading the manual, but by trying things out and exploring the software. Because of this, some of the features of a piece of software are likely never to be discovered by users if they are hidden inside a number of levels of menus or option dialogs. There is a compromise to be reached, trading off the ease with which a user can find features against the possibility of presenting them with too much information and too many options, causing them to become confused or daunted by the application's interface. Jeff Raskin discusses such issues at length in his book The Humane Interface[21].

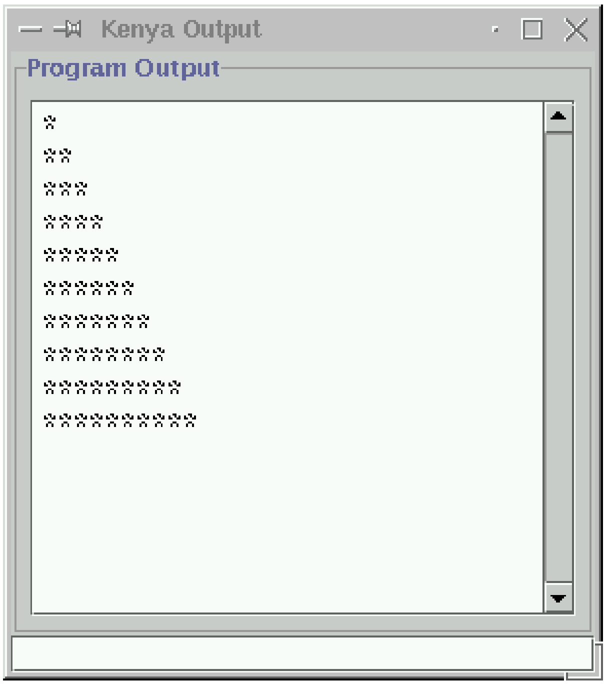
In the design of the Kenya system, I have tried not to include features which are unessential, and to only provide one way of doing everything. (Larry Wall, the inventor of Perl[22] has a famous adage *"There's always more than one way to do it"*. Perl is a notoriously difficult language to master.) This should make it easier for the user to learn all of the features, and never to find themselves thinking "I wonder if I'm doing this the right way, perhaps another way would be better ...". Having only a relatively small number of operations that the user can perform means that all of the features can be presented on the user interface without it becoming cluttered or confusing.

I designed the interface which is shown below to address these issues. It has been created using components from the Java Swing classes. The user can swap between editing the Kenya or Java versions of their code, which are displayed in tabbed panes. All of the operations available (opening a file, compiling the code ...) are given buttons on a tool bar. At the bottom of the window there is a box in which status messages are displayed, giving the status of program compilation etc.



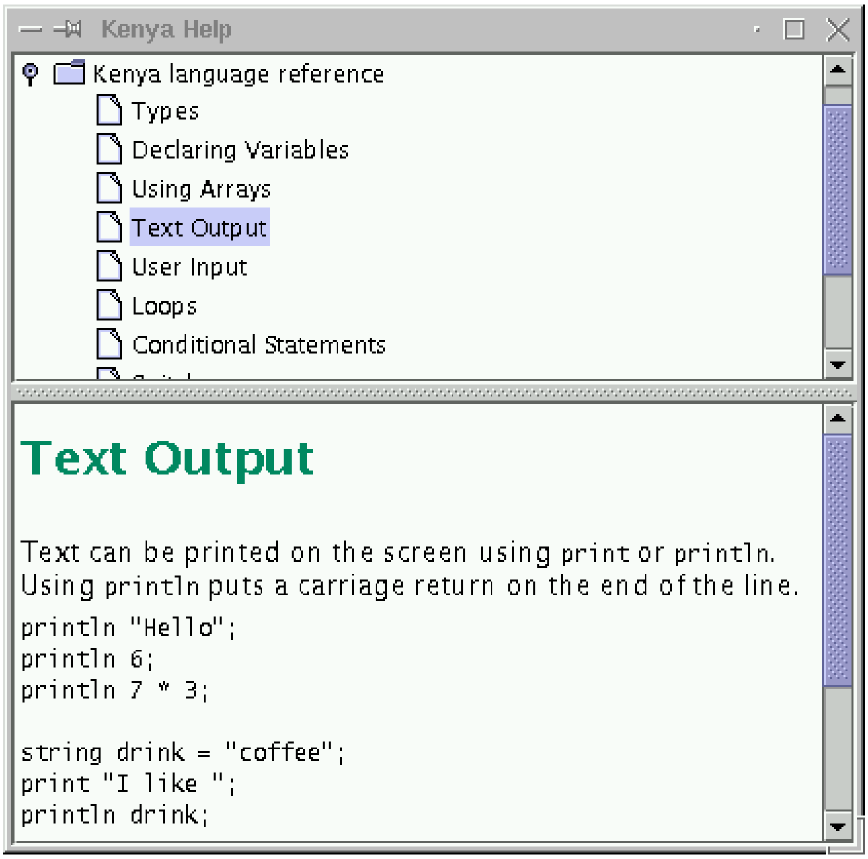
When a program is executed, another window is opened showing the output which it produces. I wanted to keep the execution of the program inside the Kenya system to give a more homogenous feel to development. When developing with something like Sun's JDK for Java, the code is edited in one program, compiled using a command line compiler, and then run either on the console or showing its own windows. This means the programmer has to switch between different windows and tools depending on what they want to do. Although this allows the more advanced programmer to use their own favourite tools for each part of the development process, customising the way that they work, it is less appropriate for a novice.

This is the output window, showing the execution of a program. The box at the bottom of the window allows data to be entered for programs that require user interaction.



Although people are less inclined to read manuals than they once were, there is still a requirement for good support documentation and this may more usefully be supplied in the form of online help. Such documentation is particularly valuable in a programming environment, as a language reference can be provided in which programmers can quickly look up syntax etc. A Java developer will be likely to keep a web browser window open on the API documentation page[23] as they work, and programmers working with Microsoft's libraries find the MDSN[24] references invaluable. As the Kenya IDE is targeted toward a specific programming language, such reference material can be provided for access inside the same interface, meaning that a network connection is not necessary to access the documentation. The only disadvantage with this is that any updates made to the documentation will be slow to propagate to users.

This is an example of the help window, where users can find references on the system and the language.



Evaluation

In this section the results of tests performed on the language and the development environment will be presented.

Testing the Language

The Kenya language was designed to be used by novice programmers to solve the sorts of problems set in introductory programming courses. Solutions to a selection of the exercises set for the first year computing laboratories[25] at Imperial College have been coded in Kenya. The code for these solutions is given in Appendix C along with their translations to Java.

The fact that solutions to these exercises can be coded in Kenya shows that it would be possible to use it for teaching a first year programming course with similar exercises.

Exercises

Star Triangle

The object of this exercise is to draw a triangle of stars (asterisks) on the screen with one star at the top and ten at the bottom. This demonstrates the use of textual output, integer variables and for loops.

Fibonacci

The object of this exercise is to calculate the nth term of a Fibonacci sequence where n and the two starting terms (positions 0 and 1) are given by the user. This demonstrates the definition and use of a recursive function, conditional statements and getting input from the user.

Put Base

The object of this exercise is to produce a program which outputs a given number in a given base. This demonstrates the use of a void function and the mathematical modulus operator.

A comparison of the amount of code which needs to be written (in terms of the number of characters) between Kenya, Turing and Java is given below. Comments were removed from all the programs before measuring their size so that only executed code was taken into account.

Exercise	Kenya	Turing	Java
Star Triangle	89 chars	242 chars	259 chars
Fibonacci	519	520	734
Put Base	400	507	641

The size of programs to solve different exercises in different languages

In all cases it can be seen that Kenya is at least as concise as either of the other languages. This is good as it points to Kenya being a simple language with no unnecessary syntax, which was one of the key design goals.

The newly designed Kenya does posess many of the qualities of a good teaching language, and fulfils the above aims to a high degree. It is of course not possible to say that Kenya is a good teaching language without it being used to teach a novice programmer or an introductory programming class. Unfortunately, at this stage in the year there are hardly any novice programmers in the department, due to the college's highly effective teaching (and the threat of exams!) and so it was difficult to test it "in the field". Also the environment has not been developed to be sufficiently robust to give to a class for laboratory work. It is intended only to demonstrate the possibilities of working with the language.

The Language Implementation

The implementation of the Kenya language produced in this project is not quite in line with the original specification. A couple of extra restrictions were added in order to fix some of the shift/reduce errors encountered while developing the grammar for the language. The grammar had to be changed at several points during the development of the translator and the type checker in order that the necessary parts of the data structure were available from particular nodes in the tree (mostly this was achieved by "flattening" productions).

With the pressure of time mounting towards the end of the project, two compromises were necessary. When a function call is made, the keyword `call` must be placed in front of the name of the function. When a logical not operator is applied, the expression to which it applies must be in brackets, even if it only consists of one term. Also, unfortunately during the development of the grammar, the unary minus operator was accidentally omitted, and so is not present in this implementation. If further development is done on an implementation of Kenya, work should be undertaken to correct the grammar so that the language can be used exactly as it was designed without these restrictions.

Apart from these points, the implementation allows full use of the language as it was designed. In all the tests carried out, the translator produced correct Java code from correct Kenya programs.

The Kenya language is designed to be as simple as possible whilst still being useful. This means that some "shortcuts" used by more experienced programmers (for example the use of the increment operator) are not provided. As the translator produces Java code directly related to the Kenya code written, the increment operator is not used in the Java code (except in the one case of modifying the index variable in a for loop). This may not be the code that a Java programmer would write, but it is clear what the code does in relation to the Kenya code.

An interesting point to consider is whether the translator should perform any optimisations. For instance in the above case the translator might come across:

```
a = a + 1;
```

in the Kenya code. The current system would write out exactly the same statement in the Java code, but Java has an increment operator, so if the translator is to follow the rule of generating the code that a human Java programmer would write then it should possibly generate the shorter:

```
a += 1;
```

or even

```
a++;
```

Taking account of these cases would add complexity to the design of the translator, as the identifier used in the right hand side of the expression needs to be matched with that on the left. This is something to be considered as a future improvement to the system.

Another optimisation which might be made is that of constant folding, i.e. if the result of an expression can be computed at compile time rather than having to be left to be computed at runtime this will save on the required computation when the program is run. This technique might be applied if the following code is encountered:

```
const int secondsInDay =  60 * 60 * 24;
const real pi = 22.0 / 7.0;
```

In both of these cases the value to be assigned to the constant can be calculated at compile time. It is difficult to say whether it should be done at translation time. In the first case, the fact that the multiplication is written out helps to assure the programmer that the value is correct. In the second case, it is more debatable. Most compilers will do constant folding when generating executable code, and so there should not be a speed increase associated with calculation of the value at translation time. However, if the code was:

```
real diameter = 23.7;
real radius = diameter / 2;
```

then not all of the operands of the division are real numbers, and a type conversion needs to be applied to the 2 before the calculation can be done. The compiler will either add in such a function so that the expression becomes something like

```
radius = diameter / int_to_real( 2 );
```

or it will rewrite the value 2 as 2.0 . Perhaps this should be done at translation time, as the division of a real by an integer is a type mismatch, and maybe the programmer should know if the compiler is going to have to do something to fix their code so that it works (especially if it has a computational cost, as with a function call). This is not done in the current Kenya system, and it just allows through arithmetic with mixed integer and real terms and assumes the compiler will deal with it.

Testing the Development Environment

The software developed in this project provides a development environment for writing Kenya programs, checking their correctness and translating them into Java. This tool was written in Java, and the code comprises just over 30,000 lines of source code. Approximately 20,000 lines of this code was generated by SableCC to create the lexical analyser, the parser and the classes which represent the different nodes in the abstract syntax tree.

Responsiveness

As the code for this tool is quite large, I decided not to create all of the objects when the application starts up, but to put off creating objects and loading classes until they were needed (load on first use). This is possible as Java supports dynamic class loading. Doing this decreases the amount of time that the user has to wait when starting the application. It does mean, however, that the user will experience a short delay the first time that they activate a certain feature, for instance the first time they parse a piece of code or access the help. This delay does not occur on repeated use of these features as the classes have already been loaded and initialised. In no case is the delay more than a second or two, and this is deemed acceptable for general use.

Running the Kenya system on a machine with a 300MHz Pentium II processor, Redhat Linux v6.2 and the IBM Java Virtual Machine, the time taken to parse, type check and translate one of the programs from the exercises given above averages approximately 0.5 seconds. This time is much smaller than the time taken to compile the Java code, which averages around 5 seconds. The delay added to producing running code by having to translate it from Kenya to Java is therefore negligible.

Usability

Several students of the Department of Computing at Imperial College were given the system to test. (These students were actually quite experienced programmers, rather than novices, but their experience was considered valuable in assessing the usability of the system). They were given the user manual and the set of example programs which are included in the appendices of this report. None of the students experienced any difficulty in writing a simple program in Kenya, translating it to Java, compiling and running it.

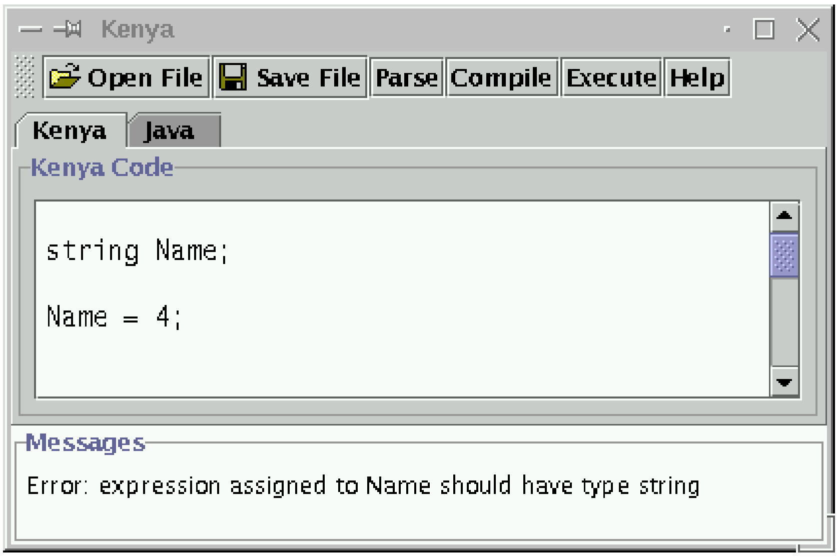
The error messages produced by the type checker (discussed further below) seemed sufficient for the users to identify and correct simple errors in their programs. It was mentioned that it would be nice to have the location of the error highlighted in the editing window.

The view was expressed that it would be nice to have automatic indentation in the editing windows, and that indenting the Java code in a different way might make its function clearer. These are slightly more important than just cosmetic changes, but indentation is quite personal to individual programmers.

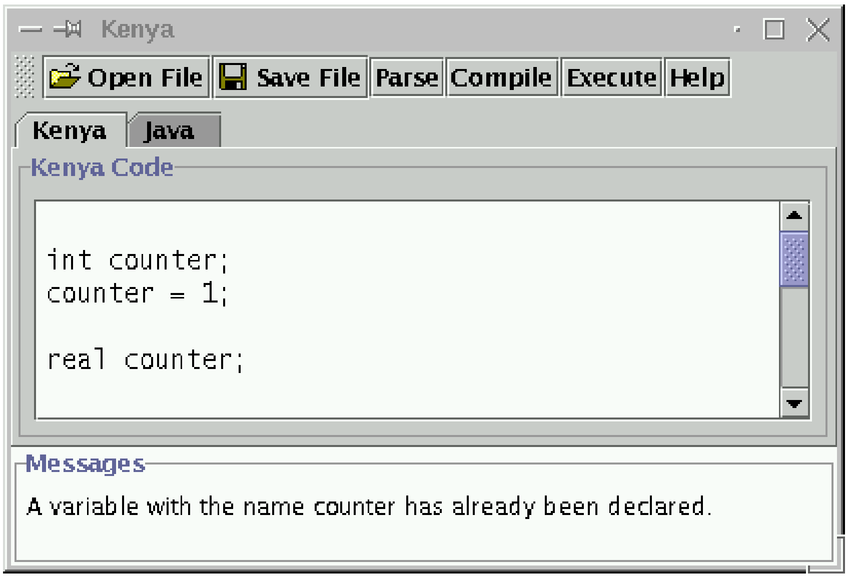
Some concerns were raised over consistency between representations of the program in the system. If the Kenya code is edited does it invalidate then Java version? If the execute button is pressed, which version of the code is run? These issues would need to be addressed before deploying the system in a laboratory environment.

Error Reporting

Some examples of errors detected by the type checker are shown below. In the first case a variable called "Name" has been declared as a string, but has had an integer assigned to it. This is an example of a type check.



In the next example, two variables are declared with the same name. This is not allowed as the machine will not know which of the two variables the programmer is referring to. This is an example of a uniqueness check.



The information that the system provides about errors in programs is probably not sufficiently detailed at the moment to enable novice programmers to debug their programs easily and fully. The lexical analyser and parser used in this project are generated automatically from the grammar. Although this does not mean that it is not possible to change the way that they report errors, if changes are made they are overwritten the next time that the parser generator is run (this needs to be done every time that a change is made to the grammar). Once the grammar has been completely finalised and there is no need to regenerate the analysis code, changes could be made to provide more friendly error messages.

One of the major failings in terms of debugging messages is that it has not been possible to report the line numbers of semantic errors. When the parser creates the abstract syntax tree representing the program it does not record in each node the line number on which the production which generated it was found. The data stored in a Node could be altered, and the parser changed to record line numbers as the code is analysed, but this would require fairly large changes to the parser. Again this would only be possible after the parser had been generated for the last time, and would be difficult as the parser was designed by someone else, and so some significant reverse engineering would be required.

Conclusions

In this section the results of the project will be reviewed to see how far they fulfil the initial aims of the work.

The overall goals of the project were:

- To examine the teaching of programming to first year students
- To consider the qualities of a language and a technique for teaching programming to novices
- To design a language having such qualities
- To implement an environment for the development of programs in this language

Examination of Teaching Languages/Techniques

In considering the way in which programming is taught to students at the beginning of their Computing degree, it was noted that most important is that the students learn the fundamental principles of program design and think clearly about the techniques and algorithms that they are using to solve problems. They should not be hindered unnecessarily by having to contend with complicated syntax in the language they are using.

At the same time, it was recognised that students often become frustrated by being made to learn what they regard as "toy" languages with no real commercial application. In view of this, in designing an approach for teaching programming, it must be taken into consideration how easy it is to progress from the teaching language to other, more sophisticated and commericially oriented languages. The majority of students studying Computing at Imperial College consider Java to be the language that they should be being taught in order for them to be employable Software Engineers on completion of their education.

This examination, along with consideration of the system currently being used to teach programming to first year students at Imperial College, revealed that there were definitely some improvements which could be made in order to increase the ease with which a "commercial" language might be learnt whilst maintaining a focus on allowing the fundamental principles of programming to be taught. These are essential if the student is to develop into a good programmer who can work with a number of different languages to express these underlying techniques and ideas.

Language Design

The strengths and weaknesses of available teaching languages were examined in detail and the qualities which would be desirable in a new language defined. The Kenya language was designed trying to incorporate as many of these features as possible, and leaving out anything that seemed unduly complex or confusing for the novice programmer. The other main consideration in designing the language was to maintain a similarity with Java where the Java syntax seemed sufficiently clear and simple. When the student moves on to programming in Java at a later stage, they should not have to learn a completely new set of keywords and syntax.

Tests were carried out to determine whether it is possible to write programs in Kenya to solve the sorts of problems given to students in an introductory programming class. Although it was not easy to test how useful Kenya is in teaching first year programmers (or fourth year programmers impersonating first year programmers) to solve such problems, example programs demonstrate that it is possible to solve the problems using the language.

Translation

As well as a language design, an approach for teaching with this language was developed. This approach is to provide an automatic translation of the Kenya program written by the student into a Java program which performs the same function. The goals for the Java code generated by the system were that it should:

- be correct
- be clear and clean
- be self-contained
- not appear too "mechanical", i.e. it should appear as "human generated" as possible; no redundant code; no extra semicolons, brackets or braces

For all the cases tested, the translation of a valid Kenya program produced valid Java code which compiled and performed the intended function of the Kenya program. However, "proof by example" is not a rigorous proof that all valid Kenya programs will be translated to valid Java programs by the software.

To verify that this is indeed the case would require specifying the operational semantics of both Kenya and Java, and showing that both programs result in the same changes of state. Unfortunately I currently have neither the skills, knowledge nor time to undertake such an exercise.

Clarity and cleanliness of code is a somewhat subjective thing. There are varying opinions on what is considered to be good code. A very experienced programmer may prefer code that is concise, and therefore quicker for them to read and appreciate. Other programmers may prefer code which provides a more verbose description of what is going on.

All the Java code that is generated by the Kenya system is self contained (i.e. requires no extra classes to be imported) apart from the code generated from input functions like `readInt()`.

The Development Environment

Novice programmers cannot be expected to work effectively with command line tools. Their learning experience will be greatly enhanced if they are provided with a development environment which helps them to write programs effectively. A development environment for Kenya was designed and implemented as part of this project. The main goals for the design of the environment were:

- it should provide everything the programmer needs to write and test programs
- it should provide as much information as possible to users about problems with their code
- it should be easy to use
- it should not provide excessive or insufficient functionality

In terms of the basic key components of a development environment, the Kenya environment provides an editing area for writing code, and a facility to compile code (after Kenya code has been translated to Java) and execute it all inside the same environment.

The code editor currently lacks features such as syntax highlighting (automatically colouring keywords to distinguish them from variable and type names etc) and auto indenting. These would be nice features to have if the editor was going to be used for serious programming as they make it much easier to see the form and structure of the code which is being worked on. The environment developed in this project was intended to demonstrate that development with the Kenya language is possible. The fact that these features are missing at this stage is not a major failing, but they should be added if further work is done on the project.

Other features which might be seen as being missing are the ability to load and save Java code (rather than Kenya code) or to change the package name after it has been set. There will inevitably be extra features that people will want added to a development environment, and these could be worked on further in future.

The type checker works quite well, and will detect whether the type of the expression on the right hand side of an assignment matches the type of the variable to which the assignment is being made. It also reports the use of types which have not been defined, and the repeated use of type or variable names.

Currently any compiler messages generated by the GJ compiler are passed straight back to the user without being processed. If it were possible to add line number tags, then it would be much better if the line number reported in a compiler error could be matched to the line of Java which caused it, and then that line be mapped back to the line of Kenya which generated it. This would associate any compiler errors directly with the Kenya. This would be better than the current situation, as Kenya users should not have to concern themselves with the Java stage unless they wish to.

Overall this project has yielded some insights into various aspects of teaching and learning programming. A language and an initial version of a development environment have been designed and implemented. Some areas have been identified on which further work would need to be done before the language could be used in a laboratory environment. These are detailed in the next section.

Further Work

There are a number of extensions which could be made to this project. These fall broadly into two categories: small, incremental improvements to the development environment, and larger developments, which might well constitute a project in their own right.

Small Developments

- **Improvements to the robustness of the IDE -** if Kenya were to be used in a classroom environment, more work would have to be done to make sure that it stood up to the rigours of laboratory use.
- **The provision of file handling in Kenya -** file handling is a fairly fundamental thing to want to do, allowing programs to be run with larger amounts of data than can be easily input by hand.
- **The addition of graphical output to Kenya -** Java's Swing libraries make it relatively easy to produce graphical user interfaces for programs, and it might be possible to provide the same sort of functionality in Kenya.

Larger Developments

- **A step through debugger -** allowing users to step through the execution of their program one instruction at a time, and to examine the contents of variables as they do so, is very valuable when debugging. This would probably mean creating an interpreter for Kenya code rather than translating it to Java, compiling and runnning on the Java virtual machine.
- **Improvements to the error reporting -** errors are currently reported in terms of unexpected tokens, incorrect types etc. It would be interesting to try and provide error reporting which identified the cause of the problem, not just the symptoms.

References

1. Sun Microsystems, *The Source for Java(TM) Technology* http://java.sun.com

2. Holtsoft, *Turing and Object-Oriented Turing homepage* http://www.holtsoft.com/turing

3. Microsoft homepage http://www.microsoft.com

4. Lam, Y.K. *IDE for Beginning Java Programmers* 2000

5. BlueJ, *BlueJ - Teaching Java* http://www.bluej.org

6. Motil, J. and Epstein D. *JJ : a Language Designed for Beginners (Less Is More)* available at http://www.publicstaticvoidmain.com

7. Bailey, R., *Language Processors* 1998, Imperial College

8. Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers - Principles, Techniques and Tools* 1986 Addison Wesley.

9. *The Lex and Yacc page* http://www.combo.org/lex_yacc_page/

10. JGuru *ANTLR website* http://www.antlr.org

11. Moog, T. *PCCTS resources and Notes for New Users* http://www.polhode.com/pccts.html

12. Sable Research Group *SableCC 2.16.2 Homepage* http://www.sablecc.org

13. Gamma et al *Design Patterns* 1995, Addison Wesley

14. Jackson, C. and Tomlins, A. *Java Generics* 1999, http://www.doc.ic.ac.uk/~caj97/projects/course/java-generics/report

15. Java Developer Connection *Prototype for Adding Generics to the Java(TM) Programming Language* http://developer.java.sun.com/developer/earlyAccess/adding_generics/

16. Bracha, G., Odersky, M. , Stoutamire, D. and Wadler, P. *GJ, extending the Java language with type parameters* 1998

17. Pooley, R. *Extended Bachus Naur Form* http://www.cee.hw.ac.uk/~rjp/Coursewww/Cwww/EBNF.html

18. Eckel, B. *Thinking In Java (2nd Edition)* 2000, Prentice Hall

19. Microsoft *Visual Studio* http://msdn.microsoft.com/vstudio/

20. Sun Microsystems *Forte* http://www.sun.com/forte/ffj/overview.html

21. Raskin, J. *The Humane Interface* 2000, Addison Wesley

22. Wall, L. et al *Programming Perl* 2000, O'Reilly

23. Sun Microsystems *Java 2 Platform SE v1.3* http://java.sun.com/j2se/1.3/docs/api/

24. Microsoft *MSDN Online Library* http://msdn.microsoft.com/library/

25. Imperial College Dept of Computing 1st year lab exercises
http://www.doc.ic.ac.uk/lab/cs1/lab_specs.html

Appendix A - A Grammar for Kenya

The grammar for Kenya for use with SableCC looks like:

```
Package minijava;

Helpers
  letter = ['A'..'Z'] | ['a'..'z'];
  digit = ['0'..'9'];
  cr = 13;
  lf = 10;
  point = '.';
  not_cr_lf = [[32..127] - [cr + lf]];
  space = ' ';

Tokens
  boolean = 'boolean';
  char = 'char';
  int = 'int';
  real = 'real';
  string = 'string';
  void = 'void';

  klass = 'class';
  const = 'const';
  print = 'print';
  println = 'println';
  if = 'if';
  else = 'else';
  while = 'while';
  return = 'return';
  switch = 'switch';
  case = 'case';
  break = 'break';
  default = 'default';
  for = 'for';
  to = 'to';
  step = 'step';
  decreasing = 'decreasing';
  call = 'call';

  true = 'true';
  false = 'false';

  and = 'and';
  or = 'or';
  xor = 'xor';
  not = 'not';
```

```
  readint = 'readInt()';
  readreal = 'readReal()';
  readstring = 'readString()';

  identifier = letter (letter | digit)*;
  stringliteral = '"' [not_cr_lf - '"']* '"';

  l_parenthese = '(';
  r_parenthese = ')';
  l_brace = '{';
  r_brace = '}';
  l_bracket = '[';
  r_bracket = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  dot = point;

  plus = '+';
  minus = '-';
  times = '*';
  divide = '/';
  power = '^';
  mod = '%';
  less = '<';
  lessequal = '<=';
  greater = '>';
  greaterequal = '>=';
  equal = '==';
  notequal = '!=';
  assign = '=';

  intnumber = digit+ ;
  fpnumber = digit+ point digit+;

  new_line = cr | lf | cr lf;

  blank = space*;

  comment = '//' (letter | digit | space )* ( cr | lf | cr lf );

Ignored Tokens
  blank , new_line , comment;

Productions

 statements =
    {list}  statement statements |
    {empty} ;
```

```
  statement =
    {dec} declaration |
    {functioncall} function_application semicolon |
    {assignment} field_access assign expression semicolon |
    {if} if l_parenthese bool_expression r_parenthese
[block1]:block else? [block2]:block? |
    {while} while l_parenthese bool_expression r_parenthese
block |
    {return} return expression? semicolon |
    {switch} switch l_parenthese expression r_parenthese
switch_block |
    {for} for decreasing? name assign [exp1]:expression to
[exp2]:expression step? [exp3]:expression? block |
    {break} break semicolon |
    {print_str} print expression semicolon |
    {println_str} println expression semicolon;

  block = l_brace statements r_brace;

function_application = call name l_parenthese actual_param_list
     r_parenthese;

actual_param_list = {list} actual_param_list comma expression |
                    {exp} expression |
                    {empty};

  switch_block = l_brace possible_case* r_brace;

  possible_case = switch_label block;

  switch_label =
    {case} case expression colon |
    {default} default colon;

  declaration =
    {class_dec} klass identifier l_brace declaration* r_brace |
    {func_dec}  type identifier l_parenthese formal_param_list?
r_parenthese block |
    {const_dec} const type identifier initialiser? semicolon |
    {var_dec}   type type_param? identifier array_access*
initialiser? semicolon;

  formal_param_list = type_name comma_type_name*;

  type_name = type identifier;

  comma_type_name = comma type_name;

  initialiser = assign expression;

  booleanliteral =
    {true} true |
    {false} false;
```

```
/********************
       Types
*******************/

  type =

    {basic_type}
        basic_type |

    {reference_type}
        reference_type;

   basic_type =

    {char}
        char |

    {int}
      int |

    {real}
      real |

    {string}
        string |

    {boolean}
        boolean |

    {void}
        void;

  reference_type =
    {class_type}
        class_type;

/********************
Variable Declarations
*******************/

  type_param = less type_param_list greater;

  type_param_list = type comma_type*;

  comma_type = comma type;

/********************
  Expressions
*******************/

  expression =
    {boolexp} bool_expression;
```

```
  math_expression =
    {term} term |
    {plus} math_expression plus term |
    {minus} math_expression minus term ;

  term =
    {factor} factor |
    {mult} term times factor |
    {div} term divide factor |
    {mod} term mod factor |
    {power} term power factor;


  factor =
    {function} function_application |
    {number} number |
    {stringliteral} stringliteral |
    {readint} readint |
    {readreal} readreal |
    {readstring} readstring |
    {fieldaccess} field_access |
    {expression} l_parenthese bool_expression r_parenthese;

  number = {i} intnumber |
           {f} fpnumber;

  bool_expression =
    {term} bool_term |
    {mathexp} math_expression |
    {or} bool_expression or bool_term;

  bool_term =
    {factor} bool_factor |
    {and} bool_term and bool_factor;

  bool_factor =
    {literal} booleanliteral |
    {compare} [lhs]:math_expression comp_operator
[rhs]:math_expression |
    {negate}  not l_parenthese bool_expression r_parenthese;


/********************
       Names
********************/

  field_access =
    {array} name array_access+ |
    {scalar} name;
```

```
name =

  {simple_name}
      simple_name |

  {qualified_name}
      qualified_name;

simple_name =
  identifier;

qualified_name =
  field_access dot identifier;

array_access = l_bracket math_expression r_bracket;
```

Appendix B - Some Example Kenya Programs and Their Translations to Java

Hello World

Kenya

```
print "Hello World!";
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {
     System.out.print( "Hello World!" );
  }
}
```

Using Variables

Kenya

```
int a = 2;
int b = 3;
println a + b;

string h = "Hello";
println h;
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {

    int a = 2;
    int b = 3;
    System.out.println( a + b );
    String h = "Hello" ;
    System.out.println( h );
  }
}
```

Using Arrays

Kenya

```
int a[10];
a[1] = 5;
println a[1];

string b[2][2];

//arrays are indexed from zero
b[0][0] = "Hello";
println b[0][0];
```

Java

```
package examples;

public class Program {

  public static void main(String[] args)  {

    int [] a = new int [10];
    a[1] = 5;
    System.out.println( a[1] );
    String [][] b = new String [2][2];
    b[0][0] = "Hello" ;
    System.out.println( b[0][0] );
  }
}
```

Using a While Loop

Kenya

```
int i = 1;

while ( i <= 10 )
{
  println i;
  i = i + 1;
}
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {

    int i = 1;
    while ( i <= 10 )
    {
      System.out.println( i );
      i = i + 1;
    }
  }
}
```

Using a For Loop

Kenya

```
int i;

for i = 1 to 10
{
  println i;
}

for i = 2 to 10 step 2
{
  println i;
}

for decreasing i = 10 to 2
{
  println i;
}
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {

    int i;

    for( i = 1; i <= 10; i ++ )
    {
      System.out.println( i );
    }

    for( i = 2; i <= 10; i += 2 )
    {
      System.out.println( i );
    }

    for( i = 10; i >= 2; i-- )
    {
      System.out.println( i );
    }
  }
}
```

Using a User Defined Data Type

Kenya

```
class Person
{
  string name;
  int age;
}

Person me;
me.name = "Robert";
me.age = 22;

println "My name is " + me.name;
println "I am " + me.age + " years old";
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {
    Person me = new Person();
    me.name = "Robert";
    me.age = 22;
    System.out.println( "My name is " + me.name );
    System.out.println( "I am " + me.age + " years old" );
  }
}

class Person
{
  String name;
  int age;
}
```

Using a Conditional

Kenya

```
int a = 2;

if ( a == 2 )
{
  println "a equals 2";
}
else
{
  println "a does not equal 2";
}
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {

    int a = 2;

    if ( a == 2 )
    {
      System.out.println( "a equals 2" );
    }
    else
    {
      System.out.println( "a does not equal 2" );
    }
  }
}
```

Using a Switch

```
int a = 2;

switch ( a )
{
  case 1  : { print "One";  break; }
  case 2  : { print "Two";  break; }
  default : { print "Many"; break; }
}
```

Java

```java
package examples;

public class Program {

  public static void main(String[] args) {

    int a = 2;
    switch ( a ) {
      case 1:
      {
        System.out.print( "One" );
        break;
      }
      case 2:
      {
        System.out.print( "Two" );
        break;
      }
      default :
      {
        System.out.print( "Many" );
        break;
      }
    }
  }
}
```

Defining and Applying a Function

Kenya

```
// The definition

int triple( int n )
{
  return n*3;
}

// The application

int a = 2;
int b = call triple( a );
```

Java

```
package examples;

public class Program {

  public static void main(String[] args) {

    int a = 2;
    int b = triple( a);
    }

  static int triple( int n ) {
    return n * 3;
  }
}
```

Appendix C - More Complex Example Kenya Programs and Their Translations to Java

Star Triangle

This program prints a triangle of stars on the screen.

Kenya

```
int a;
int b;

for a = 1 to 10
{
  for b = 1 to a
  {
    print "*";
  }

  println "";
}
```

Java

```
package star;

public class Program {

  public static void main(String[] args)  {

    int a;
    int b;
    for( a = 1; a <= 10; a ++ )
    {

      for( b = 1; b <= a; b ++ )
      {
        System.out.print( "*"  );
      }

      System.out.println( ""  );
    }
  }
}
```

Fibonacci

This program generates the nth number in the Fibonacci sequence starting with the given terms.

Kenya

```
int fibonacci( int first, int second,int N )
{
   if ( N == 0 ) {
     return first;
   }
   if ( N == 1 ) {
     return second;
   }
   else {
     return call fibonacci( second, first+second, N-1 );
   }
}

int number;
int first;
int second;

print "Input a Fibonacci series position: ";
number = readInt();
print "Input the first and second starting values: ";
first = readInt();
second = readInt();
print "Fibonacci number at position " + number + " in series = "
+ call fibonacci( first , second , number);
```

Java

```
package fibonacci;

import kenya.io.IntReader;

public class Program {

  public static void main(String[] args)  {

    int number;
    int first;
    int second;
    System.out.print( "Input a Fibonacci series position: "  );
    number = IntReader.read();
    System.out.print( "Input the first and second starting
values: "  );
    first = IntReader.read();
    second = IntReader.read();
```

```
    System.out.print( "Fibonacci number at position "  + number
+ " in series = "  + fibonacci( first, second, number) );
  }

  static int  fibonacci( int first , int second , int N)
  {
    if ( N == 0 )
    {
      return first;
    }
    if ( N == 1 )
    {
      return second;
    }
    else
    {
      return fibonacci( second, first + second, N - 1);
    }
  }
}
```

Put Base

This program converts a given decimal number to the given base (1 - 10).

Kenya

```
void putBase( int number , int base )
{
  if ( number < base ) { print number; }
  else {
    call putBase( number / base , base );
    println " " + number % base;
  }
}

int number;
int base;

print "Intput an integer in decimal: ";
number = readInt();
print "Input the base you want it expressed in: ";
base = readInt();
print number + " in base " + base + " is: ";
call putBase( number, base );
```

Java

```
package putbase;

import kenya.io.IntReader;

public class Program {

  public static void main(String[] args)  {

    int number;
    int base;
    System.out.print( "Intput an integer in decimal: "  );
    number = IntReader.read();
    System.out.print( "Input the base you want it expressed in: 
"  );
    base = IntReader.read();
    System.out.print( number + " in base "  + base + " is: "  );
    putBase( number, base);
  }

  static void  putBase( int number , int base)
  {
    if ( number < base )
    {
```

```
      System.out.print( number );
    }
    else
    {
      putBase( number / base, base);
      System.out.println( " "  + number % base );
    }
  }
}
```

Appendix D – A Survey of People's Experiences of Learning Programming

To find out which features of a language make it easy to learn programming with, a survey was conducted of undergraduate students at different universities to discover their experiences of being taught programming.

A student from the University of Durham wrote *"I'd hate to be thrown straight into object-oriented programming."* It can be argued that object-oriented programming is a viable introductory programming style, depending on the method by which it is taught. To make this an effective method would probably involve any actual programming being preceded by a course in object-oriented design to give a good grounding in the principles of the structure of an object-oriented piece of software.

A student from University College London wrote *"I think the best way of teaching modern languages is by showing them an abstract class diagram which is directly connected to whatever programming statements the user types in"*. He advocates the object-oriented approach to programming, but with the caveat that *"pointers and references get very mind-bending at times!"* This perhaps presents a case for structured programming or the use of a language closer to Java than to C++, Java not having explicit pointer types, therefore avoiding some of the confusion caused by the subtleties of pointers and references.

A student from the University of Edinburgh wrote *"I actually learned to program with Basic"* ... *"What made Basic so understandable was the very simply syntax of everything: there were no braces, no special characters at the end of lines, and the commands did pretty much what you'd expect (you have to agree* `PRINT "HELLO"` *is a bit more intuitive then* `printf("Hello\n");`*)."* This points out the need for an introductory language to have a simple syntax so that it is not difficult to understand, and the programmer can concentrate on their algorithm.

A student from the University of Nottingham wrote *"The best* [language for teaching programming] *is Visual Basic, I think, due to the intellisense (it finishes it for you, so you don't have to remember every object's property names or methods to the letter), it checks syntax as* *you go ... you can build GUIs nice and quick which look good with an intrinsically integrated IDE with good debugging features."*. The merits of endorsing a Microsoft[3] product will not be argued here. It is true that the Intellisense feature is an aid to programmers, but it is a feature of the development environment rather than the language. The same can be said of good debugging features and any sort of graphical builder. Generating incomprehensible code from mouse clicks is probably not desirable in teaching programming.

Kenya User Guide

System Requirements

Currently Kenya only runs on Linux. As it is written entirely in Java, a conversion to other platforms should be relatively easy, and may be done in the future.

In order to take advantage of generics, Kenya uses the GJ compiler rather than the standard Java compiler. This needs to be installed in order to use Kenya.

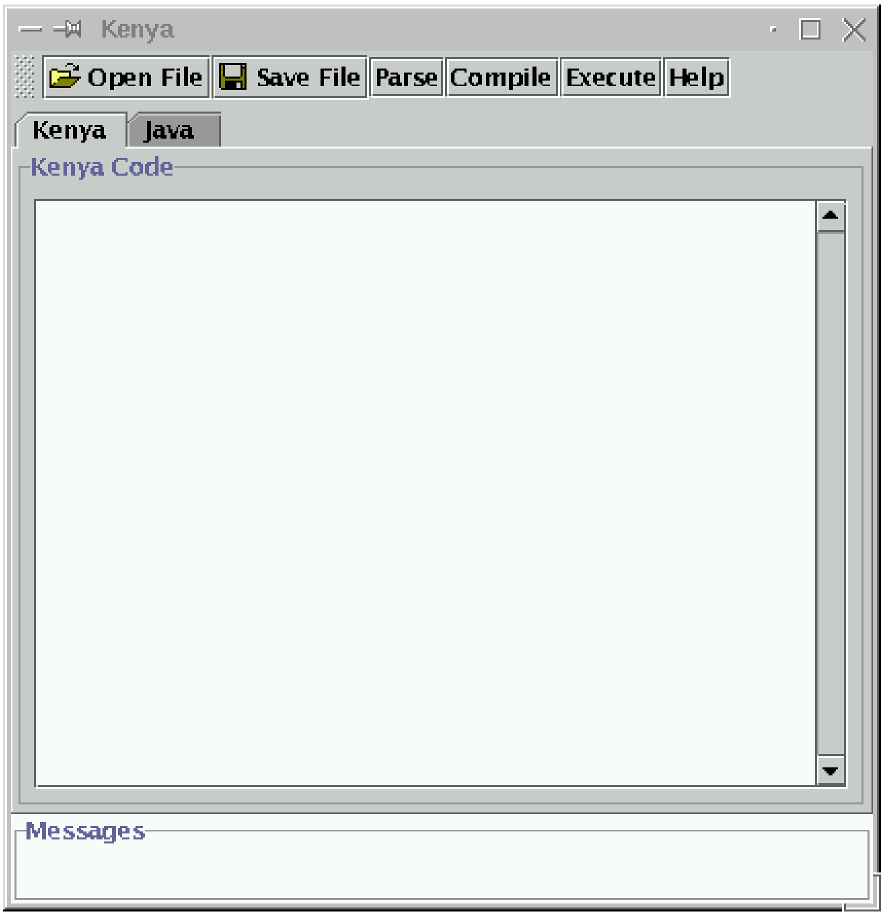
Starting Kenya

To start the system, ensure that the directory in which Kenya is installed is in your classpath and type

```
java Kenya
```

The Main Window

The main window looks like this:



The buttons in the toolbar allow you to do the following:

- Open a Kenya file
- Save a Kenya file
- Parse Kenya code to generate Java
- Compile Java code
- Execute compiled Java code
- Open the online help

In the middle of the window are two tabbed editing areas, one for Kenya and one for Java. You can switch between the areas by clicking on the tabs at the top.

The messages box at the bottom is where any messages will be displayed about errors in your code, compiler messages etc.

Your First Kenya Program

Traditionally the first program that people write in any language is the Hello World program. This prints out Hello World! on the screen. Lets see how to write and run this in Kenya.

First make sure that the Kenya editing area is selected, and type into it the following code:

```
print "Hello World!";
```

Now click on the **Parse** button. You will be asked to enter a *package name*. A package is what Java uses to keep all of the files associated with a particular program together. Enter the name HelloWorld into the box and click **ok**.

After a second or so, the display should change to show the Java area containing the Java code which the system has generated. Now click on the **Compile** button and after a few seconds you should see a "Compiled OK" message box. Click **ok** in this window.

You can now run the program by clicking the **Execute** button. An output window will pop up and you should see the words Hello World! in the window.

Close the window and return to the Kenya code. You can now save your Kenya program by clicking on the **Save** button. You will be presented with a standard file selection dialog box. Choose a suitable directory to save the file in, type the name HelloWorld.k into the selection box and press return.