CS4997 – Undergraduate Honours Thesis

CFlat: An Improved C++

By

Adam Richard

Supervisor: Bradford G. Nickerson

University Of New Brunswick, Faculty Of Computer Science

Due: 12 August 2004

Abstract

This thesis presents a new programming language called CFlat. CFlat is based on C++, and is meant to improve on C++ by making it more powerful and easier to use.

CFlat is not backwards compatible with either C or C++, but it is similar enough to C++ that a program could be easily written to translate C++ to CFlat. The design philosophy followed here is to remove redundant and/or unnecessary features and have C++ code translated to CFlat, rather than base the language on existing code. The final result is both a simpler compiler and simpler code, which results in fewer bugs.

The thesis report describes the 31 features of CFlat that differ from C++. Thus, knowledge of C++ is a prerequisite of fully understanding this thesis. The pros and cons of each feature are discussed in light of the current literature, and the decisions to add to or change C++ are explained with the principle design goals of simplicity and power in mind.

For the experimental component of the thesis, I have written a simple compiler, using lex and yacc, to implement 16 of these 31 features. The features implemented have been biased toward scientific/data processing style programming (although they are also useful in other programming domains). The object-oriented features of CFlat are left out to keep the compiler simple. The compiler translates partial CFlat to C++, which can be compiled to an executable program using an existing C++ compiler (gcc was used for testing). Six small test programs, totaling 131 lines of code, were written to test the compiler. The programs test the array processing, for loops, select statement, and I/O, as well as compile-time error detection.

Table Of Contents

List Of Tables

List Of Figures

1. Introduction

This document describes a new programming language called CFlat. CFlat is based on C++, but is meant to be more powerful, more flexible, simpler, easier to use, and safer. Many people believe that the first two criteria contradict the third and fourth. Hopefully, after reading this document, they will believe otherwise.

CFlat is not backwards-compatible with either C++ or C. That is, C++ code is not necessarily CFlat code. I believe it is better, in the long run at least, to make a better language than could be made if that restriction of backwards compatibility were in place.

The previous note need not imply that programmers wanting to migrate from C++ to CFlat will need to rewrite all their code. The languages are meant to be similar; therefore, it should not be a great task to write a compiler (if it may be called that) to convert C++ code to CFlat. In fact, I intend the CFlat standard library to be almost identical to the C++ Standard Template Library that programmers have become accustomed to (the language differences would warrant a few changes, as the reader should soon see). This conversion program's first use and test could be to convert the C++ standard library to CFlat.

2. The Design Of CFlat

2.1. Design Philosophy

The purpose of a programming language is to act as an intermediary between the human (programmer) and the computer's native language. As such, humans reading code in a programming language should be able to easily understand it, almost as if they were reading a novel. Programmers writing code in it should be able to quickly know how to perform a certain task in the language, without having to deal with low-level details unless they choose to. Tasks that are conceptually simple should be simple to code.

Now, the above does not mean that the language should do everything. When I state that it should be understandable and easy to write code in, I am referring to *after* the applicable library code has been written. That is, if the language constructs are such that, after the appropriate library code is in place, a given program can be written simply and understandably, *and also* that library code is itself simple and understandable, then the language has done its job. If not, it hasn't. The job of a language, then, is to provide facilities such that libraries and the use of those libraries can be written easily and understood easily.

The goals a language should strive for, I believe, in order of importance, are as follows:

1. Power and flexibility. The programmer should be able to do a lot easily and with minimal code.
2. Simplicity / ease of programming. It should be straightforward and simple to write code, and the language shouldn't include redundant features or ones that can be

implemented just as well in library code. Of course, since power has a higher priority, simple means as simple as power will allow.

3. Safety. The compiler should discover as many errors as possible; as many as possible of the ones it misses should cause the program to crash at runtime. Very few unintentional errors should cause the program to give incorrect results.
4. Speed and size of the final executable.
5. Speed of writing code. This isn't quite the same as ease of programming in point 2. A language that lets programmers write code quickly may have a brevity to its syntax, but the final result may not be very maintainable or understandable. Ease of programming also involves ease of maintaining code (when it has been well written).

It's interesting to note that RAD (Rapid Application Development) programming languages have placed point 5 first. This may be OK for writing a small, throw-away program. But in the long term, programs written in these languages are more difficult to modify than programs written in languages where power and flexibility are the priority, and more time (and therefore money) is wasted overall.

Standard C++ mostly follows the above philosophy; besides being very powerful, it allows the programmer to write code the way he or she wants (i.e. it is “multiparadigm”, as Bjarne Stroustrup, its creator, noted [SEBESTA 2003]). Many languages, on the other hand, have tried to force programmers to use a certain programming style, seemingly in an effort to force programmers to write good code. I am of the belief, rather, that no language can prevent programmers from writing bad code if that is what they're determined to do. Rather, languages should *allow* programmers to write good code *in every case.* No language I know of does this as well as it could.

C++ has often focused on efficiency over safety. For example, it doesn't check for array-out-of-bounds errors at runtime. A good argument can be made as to why speed is more important than safety: although it takes longer to write and debug a program written in a language with these priorities, once it is finally working, the result is a faster, and therefore better, program. That approach is problematic for two reasons: first, because it is very difficult to prove that a program works correctly; testing the program thousands of times does not prove this. Second, few programs actually reach that “final state” - instead, they constantly change and evolve. This means that at any given time there is a reasonable chance that they have a bug, which is often extremely important to discover.

When I refer to safety, I mean not only catching errors at compile time, but also having the program crash rather than continue when the code is incorrect. Having a program give an incorrect result is much worse than having it crash, because the user might not even recognize the incorrect result as an error. On the other hand, if it crashes, then we know the program has a bug by definition, and the source of the bug can usually be found more easily as well.

Even despite the above, it is often a difficult tradeoff that must be made between safety and speed. In some cases it is better to focus on speed, in others it is better to focus on safety. If I were to force people to choose safety in CFlat, they would simply revert to using C for some applications (and rightly so). Therefore, in keeping with the general engineering principle that unless something should always be done a certain way, the user should have a choice, it is my policy to let the user choose. Many runtime checks – such as array indexing out of bounds, overflow errors, and asserts – are required in the executable program's by default (and cause an exception to be thrown upon failure). However, a compiler switch to turn these checks off is also required. I believe this switch should in general only be used if the programmer is very sure that there are no bugs in the program, and if the program is not safety-critical, and if there are relatively significant gains to having the program run faster than it would otherwise.

It is also important to note that often safety, meaning the infrequent appearance of unintentional bugs, often goes together with a simple and powerful interface. This is because a small amount of code tends to have far fewer bugs than a large amount of code. For example, Java, although it claims to focus on safety, has a relatively complex and confusing way of supporting input and output, which leads to programs with more bugs.

Also, if a language has subtle inconsistencies in syntax and semantics, as C and C++ often do, then the language is unsafe because programmers will accidentally write one thing when they mean another. The “=” operator is an example (see section 4.6.5). An example in Java is the fact that all objects are references and variables of primitive type aren't.

Bjarne Stroustrup wrote of the design of C++, “Simplicity was an important design criterion: where there was a choice between simplifying the language definition and simplifying the compiler, the former was chosen” [STROUSTRUP, 1997]. I completely agree with him on this statement and have made simplicity from a programmer's point of view, without the sacrifice of power, a high priority for CFlat. I strongly believe that a powerful and useful computer tool need not be complex.

Despite Stroustrup's statement above, C++ is considerably more complex than most people are comfortable with. Much of this complexity originates from its requirement of backwards compatibility with C. Since CFlat doesn't have this restriction, it should be better able to meet this goal of simplicity.

Finally, I would like to write a few words about repetitive code. A fundamental rule for maintainable code is that a piece of code should only exist in one place. If it exists in more than one place, someone might update it in some places and forget to update it in others. Thus, if there is a function used by more than one program, in C and C++ you can put the function in a separate file and have each program that uses it include the function with a #include statement. Put another way, the programmer should tell the computer only what it needs to know to make the desired program, and not more than once.

I suggest that this principle can and should be extended to even the basic features of a language. For instance, suppose we have a program that is testing whether a variable's value is one of 1, 5, or 6. If they are forced to write, because of language restrictions:

```
if (var == 1 || var == 5 || var == 6) ...
```

then the code is not as good as it could be, simply because the variable's name, “var”, exists in 3 places instead of 1. If the programmer wants to change the name of the variable, there are now more places that he or she might forget to change. This example might seem insignificant to most people, but it is not insignificant when considered along with many other such restrictions in languages. The result is that it becomes difficult to write good code. The benefits of removing the need to write any repetitive code are at least fewer bugs and simpler code. As a result, one of the design goals of CFlat is that the programmer will never need to write the same information twice.

2.2. Influence Of Other Languages

CFlat has been most heavily influenced by C++, being an extension of that language. I chose C++ as a base because, of the languages I have encountered, it comes closest to my design philosophy outlined in the previous section. Once good library code is written in C++, code can be written that does a lot in a very small space and understandably, because of features like operator overloading, object-oriented techniques and implicit casting.

Also, C++ has a feature that is very important for any statically typed language: templates. Without some kind of generic programming facility such as templates, statically typed languages lack a great deal of flexibility that a corresponding dynamically typed language would have [VOEGELE, 2003]. Since I want CFlat to be a compiled rather than an interpreted language, it is critical that it supports templates or a similar feature. Few languages besides C++ currently provide this kind of feature.

CFlat is also influenced by IDL (Interactive Data Language), made by Research Systems, Inc. I liked its matrix operations, which allow operations on whole arrays at once without the need of a loop, and I would like to see that technique in a C-like language. These types of features are also present in Matlab and other matrix-oriented languages.

I also received much inspiration and some ideas from the D Programming Language [Bright, 2002]. Although I started designing CFlat before discovering D, I discovered that some of the ideas for it are close to or the same as mine, especially its focus on simplicity.

APL, a language not widely used anymore, has had a slight influence on CFlat. Its powerful matrix constructs are similar to the array features of CFlat, and in particular its handling of inner and outer products has been the source of my ideas for array operations described in section 4.3.6. Needless to say, its character set has not been incorporated into CFlat.

I owe to Pascal its assignment operator, “:=”, which has replaced the traditional “=” operator for assignment. Pascal also used a “..” operator for subrange types, and this operator has many uses in CFlat.

Python, as well as other languages, have a language construct called generator functions. These have been incorporated into CFlat almost exactly as in Python, and are used for providing a “for each” loop as described in section 4.4.4, besides their independent usefulness.

Finally, it is worth making note of Microsoft's C#, without which CFlat would have a different name, if little else.

3. Summary Of Major Features

Table 1 shows the major features of CFlat, whether the feature has been implemented, and the equivalent C++ feature, if applicable. The "Done?" column refers to whether the feature has been implemented in the CFlat 0.1 compiler. Ten of these 22 features have been implemented. Section 5 contains more details on the CFlat 0.1 compiler.

Table 1. Summary of CFlat language features and comparison to C++

Feature	***Done?***	***Sample CFlat code***	***Equivalent C++ code***
Lang Include File	No	`#include <lang>`	`N/A`

Function Headers	No	`function sqr(double x)`
		`returns double;`
			`double sqr(double x);`

Nested Functions	No	`function f() {`
		`  function g() { ... }`
		`}`
			`N/A`

Functions as Objects	No	`function f(function`
		`g);`
			`N/A`

In-place Overloaded Functions	No	`function trim();`
		`function trim()`
		`returns String;`
			`void trim_inplace();`
			`String trim();`

Operator Overloading	No	`template <class T>`
		`operator*(T A[], T B`
		`[]);`
			`Not allowed for primitive`
			`types; must create a`
			`regular function`

User-defined operators	No	`operator div(int x,`
		`int y) returns int;`
			`Not allowed; must create a`
			`regular function`

Array literals	Yes	`f({1, 2, 3});`	`int x[] = {1, 2, 3};`
			`f(x);`

Array properties	Yes	`A.li, A.ui, A.size`	`Store size in separate`
			`variable`

Array range operator	Yes	`A[1..10] := 1;`
		`A[1..10] := B[5..14];`
		`A := {1..10};`
			`Use loops.  Eg.:`
			`for (int i = 1; i <= 10;`
			`i++)`
			`  A[i] = 1;`

N-based arrays	Yes	`int A[1..10];`	`N/A (Forced to use a 0-`
			`based array or declare one`
			`larger than you need)`

In operator	No	`if (x in {1, 6, 8})`	`if (x == 1 ||`
			`    x == 6 ||`
			`    x == 8)`

Feature	***Done?***	***Sample CFlat code***	***Equivalent C++ code***

Stack-dynamic arrays	No	`int A[size];`	`int A = new int[size];`
			`delete [] A;`

Array operations	No	`B := A * 2;`	`for (int i = 0; i < sizeof (A); i++)`
			`  B[i] = A[i] * 2;`

Select	Yes	`select (x) {`
		`case 1: ...`
		`case 2: ...`
		`}`
			`switch (x) {`
			`case 1: ... break;`
			`case 2: ... break;`
			`}`

Any Expression in Select and Switch	Yes (for select)	`select (ans) {`
		`case “Yes”: ...`
		`case “No”: ...`
		`case else: ...`
		`}`
			`if (ans == “Yes”) ...`
			`else if (ans == “No”) ...`
			`else ...`

Ranged cases	Yes	`case 1..3:`	`case 1: case 2: case 3:`
Multiple Relational Operators	No	`if (x < y < z)`	`if (x < y && y < z)`
For loops	Yes	`for (int i := 1..10)`	`for (int i = 1; i <= 10; i++)`

For each loops	No	`for (int i := each(A))`
		`for (int i := primes ())`
			`N/A`

I/O operators	Yes	`<< “Hello World”;`	`std::cout << “Hello World”;`

Input Declarations	Yes	`>> int x;`	`int x;`
			`std::cin >> x;`

4. Features In Detail

4.1. Lang Include File

Many features that should ideally be viewed as part of the language by a user are most efficiently implemented as CFlat library code. These features, though each one may be in its own include file, must be packaged together into an include file called “lang”, similar to java's java.lang package. These features include strings, array operations, and standard I/O, besides potentially many others.

The difference between CFlat's lang header and the java.lang package is that lang is *not* automatically included in every program. The reason for this is that sometimes programmers may dislike the built-in facilities and may want to write their own. If it were automatically included, they would be forced to carry around the extra baggage of those features that they don't use.

The result of this is that it is highly recommended, but not required, that every CFlat program begin with the line:

```
#include <lang>
```

Some programmers write personal libraries for use in every program they write; they would be able to simply put the above include statement in their library and they would not need any more include statements in their programs than they already need.

If the user doesn't include the lang header but uses a symbol defined in it, a good compiler will notice this fact and inform them that they probably want to add the include statement.

4.2. Functions

4.2.1. Function Headers

In CFlat, function headers have the following form:

```
function name (parameters) returns returnType
```

So, for example, to declare a “sqr” function to square a number, one would write:

```
function sqr (double x) returns double;
```

The “returns” clause can be omitted, in which case it defaults to “void”.

The reasons for this change are threefold. First, I believe it makes the code more readable. In C++, when there is a long list of declarations, a reader can spend some time trying to figure out whether something is a function declaration or an object declaration. For example, consider the following statements:

```
Point x(Point o, Point p);
Point x((Point) o, (Point) p);
```

The first statement declares a function returning a Point and taking two Points as parameters. The second creates an object x of type Point, passing o and p, casted to Points, as parameters. These are two very different things, but it is not immediately obvious which is which by looking at the code.

The other two reasons for this change are that it fits better with functions as objects and nested functions, which are discussed in the following two sections.

Finally, this new syntax works better with operators and especially with the syntax for generators, as described in sections 4.2.5 and 4.4.4 respectfully.

4.2.2. Nested Functions

In CFlat, functions can be defined within any code block. The scope of such functions is that code block rather than the entire program unit.

This feature would be difficult to implement under the C and C++ function syntax. Just as readers can struggle trying to determine whether a piece of code is a function or a variable, as described above, it is also difficult for the compiler. If C-style functions were allowed within the context of other functions, this would become even more difficult for the compiler and for compiler writers. The new function syntax makes it a trivial matter for the compiler to recognize a function declaration, whether it is in another function or not.

4.2.3. Functions As Objects

Functions, in CFlat, can be thought of as objects. Variables of type "function" can be created and given existing functions as values, then called using the regular function call syntax. The new placement of the keyword "function", as described above, makes it look very much like "function" is a type specifier for a variable declaration. There is a parallel between “int x” and “function f” and indeed, that is not far from the way they would be treated by the compiler. This provides a great deal of power while retaining a simple and understandable syntax.

A sample code snippet demonstrating is shown in Fig. 1.

```
function sqr(double x) returns double {
          return x * x;
}
function f (double) returns double := sqr;
double x := f(2);
```

Fig. 1. Functions as objects

A variable of type function simply holds a reference to a particular function. Since functions can't be modified once created, there is no need to have a distinction between functions and pointers to functions. As a result, function variables can act just like pointers, except without the extra pointer syntax (though pointers to functions do use pointer syntax in C and C++ [KERNIGHAN 1988]).

4.2.4. In-place Overloaded Functions

Motivation: Consider a string class. Suppose one of the member functions of the class is a function to trim a string - that is, to remove whitespace from the start and end of "this" string. The designer of the class is here faced with a dilemma: should the function actually trim "this" string, or should it return a trimmed copy of it, allowing the user to decide what to do with it? That is, will it be used like this:

```
s.trim();   //remove spaces from start and end of s
```

or this?:

```
cout << s.trim();   //print a trimmed version of s, but don't change s
```

The writer of a library can't easily decide which function would be more convenient for the user. Indeed, in some cases one version would be preferred while in other cases, even in the same program, the other is more convenient. What this often results in is alternate versions of functions, such as trim and trim_inplace, the latter being the version that actually modifies the string. The end result is either extra functions for a user to know about, or the user lacking knowledge of the alternate version and hence using only one version, even when doing so is inefficient.

Solution: CFlat avoids these problems by introducing what are called *in-place overloaded functions*. These are simply two functions with the same name, and the same parameter list, but only one of which returns a value. The other has a void return type. The idea is that the function returning void does a computation on an object in-place, while the other function copies the applicable object and returns a modified copy. Using the example described above, this concept is shown by implementing the trim function in Fig. 2.

```
class String {
          function trim();
          function trim() returns String {
                    String toTrim := *this;  //copy this String
                    toTrim.trim();  //trim the copy
                    return toTrim;  //return the trimmed copy
          }
};
```

Fig. 2. Demonstration of writing in-place overloaded functions

The compiler decides which function is being called based on the context. For example, if the call appears on a line by itself, as in:

```
s.trim();
```

it calls the first (the in-place) function. If it is used as an expression, as in:

```
String s2 := s.trim();
```

then the second function, the one returning a String, is called.

A possible future development of this concept is to allow overloading of functions that only differ in their return type. For example, someone could write one function that returns an int, and another function with the same name and parameters but which returns a string. The compiler would decide which function is being called based on the expected type of the expression it appears in. This is similar to type inference, which appears in functional languages such as ML [SEBESTA, 2003]. However, because this development may introduce complications in a language with implicit casting, which CFlat has, it is uncertain at this time whether it is worthwhile.

4.2.5. Operator Overloading

The operator overloading feature of C++ is one which provides a lot of power and which allows code to be very understandable. It allows types to be created which appear to the user almost like a built-in type. CFlat therefore retains and extends this feature.

The operator headers look slightly different to make them more in harmony with the new function header syntax (section 4.2.1). An example CFlat operator declaration is:

```
operator *(int A[], int B[]) returns int[];   //define matrix product
```

CFlat also allows operators to be defined between primitive types. This is not possible in C++; there, operators that are not within a class have to have at least one user-defined type as a parameter [STROUSTRUP, 1997]. Allowing primitive types is partly for purposes of implementing some primitive operators as CFlat code rather than building them into the compiler, and partly to provide extra power to the user and more orthogonality to the language. The above multiplication operator for matrix products is an example of an operator between primitive types.

In addition, in the interest of preventing the user from having to write too many operators, many of the operators are defined in terms of others in the lang include file discussed in section 4.1. Specifically, the *, +, etc. operators are defined in terms of *=, +=, etc. (* is defined in terms of *= rather than *= in terms of * for efficiency reasons; the assignment operators do calculations in place, and are therefore the more primitive operations). Also, most relational operators (>, >=, <=, == and !=) are defined in terms of <. Then, for example, if the user defines a *= operator, * can be used as well without the programmer having to define it.

Note that, since (x == y) equates to !((y < x) || (x < y)) in terms of <, its efficiency could be doubled if the user defines it manually. Therefore it is recommended to define both < and == when speed may be important. Any of these automatically defined operators can be defined manually; since the operators are templates, the user's version is always used instead.

An example implementation is shown in Fig. 3.

```
template <class T, class U>
operator >(T x, U y) returns bool {
          return y < x;
}
```

Fig. 3. > operator defined in terms of <

C++ provides something similar to this for the relational operators. There is a class called "rel_ops" in the standard library, which provides all the other relational operators once < and == are defined. Unfortunately, this class requires that the user #include <utility>, that the statement "use std::rel_ops" appears before calling the operators, and it doesn't provide any other operators besides those four relational operators. CFlat overcomes all these limitations.

4.2.6. User-Defined Operators

In CFlat users can define new unary or binary operators using the same naming rules used for identifier names. The number of parameters determines whether the operator is binary or unary. This feature allows certain functions to be called with a more understandable syntax. For example, a "mod" operator could be defined to mean the same as the "%" operator. The definition might look like:

```
operator mod(int x, int y) returns int { return x % y; }
```

Bjarne Stroustrup wrote that allowing this can lead to ambiguities – is the operator left-associative or right-associative? [STROUSTRUP, 1997]. This ambiguity is resolved in CFlat by forcing all user-defined operators to be nonassociative – that is, it is invalid to chain calls to them together, as in:

```
x mod y mod z;   //Error – mod is nonassociative
```

Instead, the user must explicitly specify the order of evaluation using brackets:

```
(x mod y) mod z;   //OK
```

4.3. Arrays

4.3.1. Array Additions

First, allow me to note a few changes in the workings of regular arrays in CFlat compared to C and C++. They will become important as they are connected to some of the other features involving arrays described later on.

Array literals in CFlat are based on array initializers in C. They are allowed most places an array is allowed, not just in an initialization. So if we have a function “sort” declared as:

```
function sort(int[] a);
```

then it can be called like so:

```
sort({1, 5, 3});
```

Java partly introduced this feature – in Java, it is necessary to prefix the array literal with “int[]” if it isn't an array initialization. Others have said there is “no good reason” not to allow array initializers in other contexts [LINDEN 113]. So for the sake of unified syntax, array literals are a set of expressions (which must have the same type) enclosed in {} and separated by commas. As we will see, they are also important for integration with some of the other features described below.

CFlat arrays are not the same as pointers, as they are in C and C++. In CFlat, an array is a *kind* of pointer – that is, it can be used wherever a pointer is allowed, but it also has other properties. A normal pointer cannot be used where an array is required. Since arrays are a separate type, it would be natural for other container types, such as those in the standard library (set, map, vector, etc.) to have an implicit cast defined from that type to the primitive array type. That way, those classes could be used in all the constructs listed below where an array is allowed.

An array has a size that it carries with it. This can be queried like so:

```
int a[] := {1, 2, 3};
cout << a.size;     //prints 3
```

It also has a lower index that can be queried (See section 3.13):

```
assert(a.li == 0);
```

and an upper index:

```
assert(a.ui == 2);
```

Note that these properties do not cause a loss of efficiency for a compiler with good optimization. If one of the properties is used in the same function that a static array is declared, the compiler can replace its use with a constant, and no extra memory is needed. If an array is the parameter to a function, the compiler may need to pass the properties as implicit parameters, but only if the function actually uses them (or calls a function which does). In those cases, the function would need that extra parameter anyway if these array properties were not available. If the properties are never used, then memory doesn't need to be allocated for it. Also, at most 2 of the 3 properties need to be stored in memory; the third can be calculated using “size == ui - li + 1”.

4.3.2. Array Range Operator

CFlat introduces a new operator, “..”. Its main use is to specify a range of values. For example,

```
1..10
```

means the integers 1 through 10 inclusive. Specifically, it can be thought of as an array containing the values 1 through 10 (it is not always treated this way by the compiler, for efficiency reasons). One use of this operator is to initialize an integer array to a range of values. For example:

```
int a[] := {1..10};
```

initializes “a” with the integers 1 through 10. The same technique can be used in an assignment statement to set an existing array to those values; an exception is thrown if the sizes of the two arrays do no match.

Arrays can be subscripted with a range to extract a slice of that array. Some examples follow.

```
double A[100], B[30] := 4.6;  //All elements of B are 4.6
A[4..20] := 3.2;   //elements 4 through 20 get the value 3.2
A[70..] := B;   //B gets copied to elements 70 and up of A
A[70..] := B[..];   //Same, but maybe more readable
```

Arrays can also be subscripted with other arrays. This allows many array elements to be assigned values without using a loop. For example:

```
int a[] := {1..10};
a[{1, 3, 5}] := a[6..8];
```

is semantically equivalent to:

```
a[1] := 6;
a[3] := 7;
a[5] := 8;
```

Since all operators can be overloaded in CFlat (see section 4.2.5), this previous feature can be implemented by defining an operator, rather than as part of the compiler. Therefore, rather than being a feature of the language, it would go in the standard library. It could be implemented by writing a function something like the one shown in Fig. 4 (it should actually be a template function so that it works for any array type, but this example is merely for demonstration).

```
operator [](int[] a, int[] indices) returns int[] {
          int retVal[indices.size];
          for (int i := 0; i < indices.size; i++) {
                    retVal[i] := a[[indices[i + indices.li]]];
          }
          return retVal;
}
```

Fig. 4. Implementing arrays as subscripts with an overloaded operator

Finally, in an assignment, an array can be used as the left hand side of the assignment while a value is on the right hand side. For example:

```
int a[10] := 5;
```

This creates an array of size 10 with every element initialized to 5.

These features will probably look familiar to programmers of Matlab or similar matrix-oriented languages. In fact, a similar set of operations, implemented through valarrays, is available in the C++ standard library [JOSUTTIS 547]. All elements in a valarray can be assigned a single value at once, and the “slice” class can be used with valarrays to extract a subset of a valarray. For example:

```
valarray<int> v(100);     //create a valarray of size 100
```

```
v = 2;          //set all elements to 2
v[slice(0, 50, 1)] = 3;  //assign elements 0 through 49 the value 3
```

In my opinion, valarrays have some design flaws; but even if an improved valarray class could be written, I believe it is better to have this functionality built directly into the language for regular arrays. The main reason is that shortcuts such as the ability to initialize all elements of an array at once are useful conceivably anywhere an array is used, not only in numerical computations. In addition, there is an advantage to having a common syntax among all the container types; it would cause programmers no end of grief if they could use shortcuts for a container in the standard library but not for ordinary arrays.

4.3.3. N-based Arrays

Whereas in C++ arrays have an implicit base of 0, arrays in CFlat can be declared with other bases. This is also done using the ".." operator and looks like this:

```
int x[1..10];
```

This above statement declares an array that can be indexed with the integers 1 through 10 inclusive. The lower index can be any integer less than or equal to the upper index. It can even be a negative integer.

The reason for this addition is that it is often counterintuitive to work with base 0 arrays. For example, imagine an array for the twelve months of the year. Most people associate February with the number 2, so would be inclined to use the index 2 when they mean February. If, because of the inflexibility of the language, they are forced to use 1 when they mean February, the possibility of countless off-by-one errors is opened up.

4.3.4. In Operator

The “in” operator is a new operator to conveniently check whether a value is one of a set of values. For example:

```
if (x in {1, 3, 4}) ...
```

The right hand side must be an array of some type X, and the left hand side must have type X (or must have a cast to X defined). The result is a boolean value that determines whether the left hand side is an element of the array on the right hand side.

Currently, this functionality would normally be written with a more complex if statement:

```
if (x == 1 || x == 3 || x == 4) ...
```

This second statement has the disadvantage that "x" must be written three times instead of one (see section 2.1, "Design Philosophy").

The “in” operator can, and is required to be, implemented in the lang include file as a template user-defined operator (user-defined operators are discussed in section 4.2.6). Since it can't be assumed that the array is sorted, it would simply do a sequential search through the array looking for the element. For this reason, the “in” operator is not very suitable for working with large arrays that are kept sorted, but is provided mainly for the convenience described above.

An alternative to this feature, which can be used in C++, would be for programmers to use a container class with a member function to do this check. This would look something like the following:

```
if (Array({1, 3, 4}).contains(x)) ...
```

This isn't quite as convenient, however. It isn't quite as readable or writable as an "in" operator. It also requires the programmer to include a separate class unless there is an “Array” class already built into the language.

4.3.5. Stack-dynamic Arrays As A Built-in Type

Motivation: In C++, static arrays (ones whose size is known to the compiler) are simple and straightforward. Dynamic arrays, on the other hand (ones whose size is determined at the time they are initialized) are not as straightforward.

First, the syntax for dynamic arrays is different than for static arrays, though conceptually the two are not much different. This may cause beginning programmers to write something like:

```
int size;
cin >> size;
int A[size];
```

and they might be confused as to why it won't compile.

Second, and probably worse, is that dynamic arrays must be deleted with a delete statement. For simple functions, this is not a big deal; but for functions with multiple return statements, or which throw exceptions, it can be tricky to ensure that the delete statement is executed in all cases. This problem is outlined in more depth by Nicolai Josuttis in the context of the auto_ptr class in section 4.2 of his book [JOSUTTIS, 1999]. This problem leads many programmers to use a less efficient container class than he or she needs, such as the vector class in the standard library, to circumvent the problem.

CFlat improves this situation by having an array class that is tied to the compiler. Whenever an array declaration occurs, the compiler checks whether its size can be computed at compile time (such as if its size is an integer constant). If it can be, it is treated as a static array, which needs no deallocation. If not, which may be the case if its size is a variable, then the compiler treats the declaration of the array as a declaration of a variable of type **`array`**. So the statement:

```
int A[size];
```

would be treated as:

```
array<int> A(size);
```

Class **`array`** is a template class defined in the lang header described in section 4.1. It does nothing but create an array with a “new” statement in the constructor, delete it in the destructor, and allow the array to be used with the regular operations for static arrays. The user can also create their own class named **`array`** and use it instead, provided it has constructors with the same interface and provided they don't #include the lang header (which would cause a duplicate definition of **`array`**).

4.3.6. Array Operations

Although the range operator operations described in section 4.3.2 are powerful, it would nonetheless be better still if programmers could write something like:

```
int A[] := {1, 2, 5, 7};
int B[] := A * 2;    //Intention is B[] := {2, 4, 10, 14};
```

This could be done with overloaded operators (see section 4.2.5), but it can bog down a programmer if he or she has to write an overloaded for each array operation he or she would like to use in an aggregate fashion. The process of writing these operators can itself be tedious and repetitive.

APL, a language no longer in wide use, had partial support for the kind of feature shown in the sample code above. It used an operator for the outer product, which, when combined with another operator, applied that operator on each combination of elements of the arrays involved. Unfortunately, this operator could only be used with the built-in operators, and not with a user defined function. [POLIVKA, 1975].

The solution in CFlat is to treat calls to functions in which arrays are passed specially. (Note that operators are nothing but functions with different syntax, so this applies to operators as well; see section 4.2.5). When a function is called, the compiler first checks for the existence of one whose parameter list matches the argument list exactly. If a match isn't found, the compiler then checks whether any of the argument is an array. If the compiler can replace a parameter of type “Array of T” with a parameter of type “T” and get a matching function, then this feature is used. It then calls that function in a loop, iterating over the elements of the array.

For example, consider the above call to the “*” operator. Suppose there is no operator defined with the header:

```
operator *(int[] A, int x) returns int[];
```

(If there was, it would simply be called and this feature would not be used). But there is a “*” operator defined which takes 2 ints and returns an int – it's built into the language. Therefore, the compiler treats the statement “A * 2” as the result of a function as defined in Fig. 5.

```
int retVal[] := A;
for (int i := A.li..A.ui)
    retVal[i] := A[i] * 2;  //Call the “*” operator to get each result
return retVal;
```

Fig. 5. Function generated by the compiler for the array operation “A * 2”

A possible future development of this idea is to extend it for use with any container class, rather than just arrays. The question that must then be asked is, what is a container? Perhaps a container could be defined as any type with an implicit cast to an array. In that case, this feature would be fine the way it is. But it might be more efficient and/or powerful to define a container as a class that provides methods, called “next”, “first” and “last”, say, to iterate through its elements. More investigation on this matter is needed, possibly including an implementation of each idea, before a decision on this can be properly made.

4.4. Program Control Constructs

4.4.1. Switch And Select Statements

The switch statement in C (which remained the same in C++) has long been a source of problems. The fact that only an integer expression may be used as the switch expression shows a lack of orthogonality. That is, the code shown in Fig. 6 makes perfect sense to a programmer reading it, but is not valid in C or its derivatives.

```
switch (answer) {
          case “yes”: ...
          case “no”: ...
          default: ... //error
}
```

Fig. 6. Invalid in C or C++ to use a string in a switch

Perhaps even worse is that once a matching case is found, execution continues through all remaining case statements in the switch, whether they match or not. This is convenient in a few cases, but in most cases it means the programmer has to add a “break” statement at the end of each case, which can be a nuisance. It also leads to bugs when the programmer forgets the break statement where one was intended, and is especially frustrating to beginning programmers. Probably the most frequent use of this “fall-through” behavior is to overcome another inflexibility in switch statements: that only one value is allowed in the case expression.

C# tried to avoid the bugs introduced by the fall-through behavior by adding the following requirement to switch statements: “Every nonempty case segment must end with an unconditional branch statement” [SEBESTA, 2003]. Whereas this is arguably a welcome change that improves the switch statement slightly, in CFlat I take more drastic measures to try to overcome all of the aforementioned limitations. This is done by making the following additions compared to C++:

1. A “select” statement. It is exactly the same as the “switch” statement, except only the *first* case that matches the select expression is executed. So a select block is equivalent to the corresponding switch block with “break;” inserted at the end of each case.

2. In both switch and select statements, any expression is valid as the switch expression, with the case statements containing expressions of the same type. So the switch statement shown above at the beginning of this section is equivalent to:

```
if (answer == “yes”) { ... }
else if (answer == “no”) { ... }
else { ... //error }
```

Note that in the switch version, “answer” only needs to be written once, whereas in the if-else block, “answer” must be written for all but one of the alternatives (see section 2.1, “Design Philosophy”).

To use switch and select, an == operator has to be defined for the type of the switch expression. It also has the unfortunate consequence that programmers might forget that to compare some types (such as C style strings), you can't use ==. However, the same sorts of problems would arise if they had to use an if-else block instead of a switch or select statement (and they have to in C++ since there switch and select can only be used with integers). So I believe the trade-off is worthwhile.

3. A range of values may be specified in a case statement. This is done using the ".." operator introduced in 3.3, as shown in Fig. 7.

```
select (num_grade) {
          case 0..49: grade := "F";
          case 50..59: grade := "D";
          case 60..69: grade := "C";
          case 70..79: grade := "B";
          case 80..89: grade := "A";
          case 90..100: grade := "A+";
}
```

Fig. 7. The .. operator in case statements

Note that where this feature is used, the type of the switch expression must be an ordinal type; that is, there must be an order to the values of that type. Specifically, the order must be defined by overloading the "<" operator, which is called to determine whether the switch expression is in the range specified in the case statement.

4.4.2. Multiple Relational Operators

In CFlat, the relational operators (==, !=, <, >, <=, and >=) can be strung together in a single expression. This is best illustrated with an example:

```
if (x < y < z) ...
```

The above statement is equivalent to:

```
if (x < y && y < z)
```

In general, any n boolean expressions may be combined with any n-1 relational operators separating them.

The advantages to this feature are twofold. First, it eliminates repetition (see section 2.1, “Design Philosophy”) because, in the above example, the expression “y” only needs to be written once, not twice. Keep in mind that in place of “y” there might be a complex expression, involving function calls, casts, etc..

Second, this notation is already valid mathematical notation that people are used to. Programmers, especially beginning programmers, are liable to write, or want to write, such an expression, because it makes sense to them. As it is now, if they did write an expression like “x < y < z” in C or C++, the code would compile without errors - “x < y” is a boolean expression evaluating to 0 or 1, and this result would be compared to check whether it is less than z. This behavior is almost never what is intended. In CFlat, these multiple relational operators in a row are parsed before the individual binary operators, so there is no ambiguity. Also, by default, ints are not implicitly converted to booleans (see section 4.6.8).

A potential concern for this feature is that it is too difficult to implement. I believe that the benefits, in the long run, outweigh this difficulty. It is *possible* to implement. The compiler, when it sees a relational operator followed by a boolean expression, checks whether another relational operator and boolean expression pair follows. If so, it adds, between the first boolean expression and second relational operator, the “&&” operator followed by another copy of the first boolean expression. The EBNF grammar would look like this:

```
full-rel-expr => bool-expr [part-rel-expr rel-op bool-expr]
part-rel-expr => rel-op bool-expr   /* bool-expr is changed to
          bool-expr && bool-expr */
```

where “bool-expr” is a boolean expression and “rel-op” is one of the six relational operators mentioned above.

4.4.3. For Loops

“For” loops in C and most languages derived from it have long been problematic. Consider a typical loop to iterate through the integers 1 to 10:

```
for (int i = 1; i <=10; i++) //loop body
```

This same loop could be instead written as a while loop, as in Fig. 8.

```
int i = 1;
while (i <= 10) {
          //loop body
          i++;
}
```

Fig. 8. A while loop being used in place of a for loop

As the reader can see, using a C-style for loop barely adds anything in the way of readability or simplicity compared to using a while loop. The only major difference is that the programmer types "for" instead of "while". Though it has been said that the for loop is "flexible" [GNU C Programming Tutorial, 2003], there is a fine line between flexibility and forcing users to handle low level details themselves. The purpose of a “for” loop is to allow the iteration through a set of values, setting a variable to each value in the set in turn; as such, the construct should make this task significantly easier to accomplish than if it weren't present. The C for loop does not.

Taking the above into consideration, CFlat introduces a new version of the “for” loop. In its most straightforward use, it might look something like the following (showing the example above):

```
for (int i := 1..10)   //loop body
```

Besides being more readable, this syntax allows the user to have to write "i" only once, rather than three times, which fits with the philosophy described in section 2.1.

A step value may also be specified, indicating the amount to increase the loop variable by each time (if omitted, it defaults to 1):

```
for (int i := 1..10 step 2)   //loop body
```

As of now, the step value must be a constant. Since negative step values are allowed to step backwards through the set of values, the compiler needs to know whether the value is positive or negative so it knows the direction of the increment. In the future, this requirement may be relaxed.

4.4.4. For Each Loops

Preliminaries: As mentioned in section 2.2, CFlat introduces generator functions. These are functions that are meant to iterate through a set of values, returning the next value in the set each time they are called. Rather than returning normally as a regular function does, generator functions save their state before returning and resume where they left off next time they are called. An example generator to iterate through the integers 1 to 10 is shown in Fig. 9.

```
function f() yields int {
          for (int i := 1..10) yield i;
}
```

Fig. 9. Sample generator function to iterate through the integers 1 to 10

The first time f is called, it returns 1, the second time it returns 2, and so on. As the reader can see, the “yield” statement is used in place of “return” to cause the function to save its state for the next call. When it reaches the end of the function, it throws an exception to indicate that it has reached the end of its set of values, then resets to its starting state for the next call.

A point worth noting is that although the set of integers above is treated as a set, the whole set is never in memory at once. So using generators allows the programmer to work with a collection of size n while only using O(1) memory (the memory required to store the state of the generator), while traditional methods often take O(n) memory.

For loop: CFlat has a new "for" loop which uses the generators described above. The alternate syntax is as follows:

```
for (variable := generator) statements
```

The loop iterates through each value returned by the generator, and stops when the generator throws its "Finished" exception. "*variable*" is assigned each of the values returned by the generator at each pass through the loop.

For each loop: Included in the lang include file is a generator called "each". This function takes an array as a parameter and yields each element of the array in turn. So the "each" generator can be used to construct a "for each" loop, as in:

```
for (int x := each(A)) x := 1;   //Assign 1 to all elements of A
```

4.5. Input And Output

4.5.1. Unary I/O Operators

CFlat introduces a convenience for doing I/O that makes it easier to do I/O from standard in and standard out. If the input/output stream is left out, it defaults to cin or cout respectively. For example, the statement:

```
<< "Hello World";
```

is equivalent to:

```
cout << “Hello World”;
```

In addition, using this new form of input/output can be done without the usual “#include <iostream>” statement, because nothing from that header is being used explicitly. This need not cause a loss of efficiency for code that doesn't do I/O, as I have demonstrated in my compiler. The parser can simply make a note of whether the unary << or >> operators are used, and if so, the code generator assumes an implied “#include <iostream>” at the beginning of the program.

4.5.2. Declarations In Input Statements

Declarations of variables may now appear in input statements. For example:

```
>> int x;
```

In C++, if you have a variable which is being initialized by an input, you have to declare it first, then use the >> operator, causing the variable's name to be written twice instead of once.

4.6. Other Minor Changes

4.6.1. Array Declarations – Square Bracket Placement

The array brackets in an array declaration can be placed either after the type or after the variable name. This allows for Java-style array declarations, which some people are used to and/or prefer. So the following is allowed:

```
int[10] x;   //Same as int x[10];
```

4.6.2. Nested Comments

Block comments (/* */ comments) can be nested in CFlat. This way, any block of code can be commented out by enclosing it in /* and */. In C++, this is not possible if the block of code already contains one of these comments, because the first */ ends the outermost comment, causing a lot of frustration.

4.6.3. Division

In C and C++, the “/” operator for division does integer division if and only if both operands are integers, and floating-point division otherwise. This can be confusing and can lead to bugs. In particular, people expect “1 / 2” to result in 0.5, and may be surprised that it's actually 0. The source of the confusion, I believe, is that the same operator is being used for what are really two different operations, integer division and floating-point division.

In CFlat, I have decided that “/” always performs floating-point division, while “\” (backslash) does integer division. One drawback to this approach is that these operators are similar and are sometimes confused – the different path separators between Windows and Unix have shown this. Even so, I believe the benefits outweigh this drawback, and that this new system is a slight improvement.

Note that even if programmers continue to use /, there won't be much of a difference in the way CFlat works compared to C and C++. Consider the following:

```
int x := 1 / 2;
```

Even under the new CFlat division rules, this will either be a compile error, or x will get the value 0, as before. If there is a cast defined from floating-point to int (which the user can define themself as the floor of the floating-point value), then the expression “1 / 2” will result in the floating-point value “0.5”, which is implicitly cast to the integer “0” by calling this function, since an integer is expected. If there is no cast defined, it's a compile error, which at least doesn't result in unexpected behavior, as can happen when dividing integers in C and C++.

4.6.4. Floating-point Constants

To make the compiler simpler to write, floating-point constants aren't quite as flexible as in C/C++. In those languages, you could end a constant with a decimal but leave off the fraction; in CFlat, you can't. So while “1.” and “1.E5” are valid floating-point constants in C and C++, they aren't in CFlat.

To continue to allow this would especially introduce difficulties due to the new “..” operator. The expression “2..5” could be, at the lexical analysis stage, either an integer followed by “..” followed by an integer, or a floating-point constant (“2.”) followed by another floating-point constant (“.5”). If the first is not allowed, there is no ambiguity and lexical analysis is simpler.

Furthermore, the only use of the extra decimal point at the end is to specify that the constant is floating-point rather than an integer. This is rarely if ever needed in CFlat, because the constant is implicitly cast to floating-point (at compile time) if necessary. It isn't needed anymore for division due to the change described in the previous section. For the few cases where the programmer wants to ensure that their number is a floating-point constant, using “2.0” instead of “2.” isn't a big deal, and it even makes more sense from a mathematical point of view.

4.6.5. Assignment Operator

In most programming languages, there is confusion as to the meaning of “=”. It can mean either assignment (giving a variable a value), or the relational operator to check whether 2 values are equal. This can lead to bugs such as the following:

```
if (x = y) ...   //programmer really meant "if (x == y) ..."
```

To fix this, in CFlat ":=" is used for assignment and "==" continues to mean comparison. "=" is not used at all, as of now.

A possible future development is that "=" could be used for either operator, letting its context decide its meaning. So if "x = 2" appears as a statement on its own, it would mean assignment (:=), whereas if it appears within an "if" statement, such as "if (x = 2)", it would mean comparison (==). This would allow programmers to continue to use "=" the way they are used to in most situations. However, this feature has its problems – in particular, "=" would now have a different meaning than in C++, and programmers might use it as if it still has the old meaning, leading to incorrect behavior. For this reason it is undecided whether this feature should eventually be incorporated into CFlat.

4.6.6. Operators For Pointers

In C and C++, accessing the member of a non-pointer is done with the "." operator, while accessing the member of a pointer is done with "->". In CFlat, both are accomplished with the "." operator, so that if "p" is a pointer, "p.m" means the same thing that "p->m" did in C and C++. The "->" operator is gone, making for a simpler syntax with fewer operators.

4.6.7. Public By Default

The default access specifier in CFlat is public. In C++ it is private [STROUSTRUP, 1997]. This includes the access specifier for inheritance, so that "class A : public B" can be more easily written "class A : B", although the former version does no harm since the "public" keyword has no effect. Making public the default allows quick-and-dirty code to be written quickly, without unexpected restrictions, while more robust code must be well thought out, which is exactly as it should be.

Since a “struct” in C++ is nothing but a class with all members public, the “struct” keyword can now be removed from CFlat as a result of this change, resulting in a simpler language.

4.6.8. Fewer Implicit Casts

In C++, there are implicit casts defined between most primitive types. Some programmers might not want this to happen because it isn't safe and might lead to unintentional bugs. Therefore, in CFlat, there is only a minimal set of implicit casts defined in the language itself. These include casting from a smaller integer type to a larger one, from a smaller floating-point type to a larger one, and from an integer type to a floating-point type. They do not include casting from floating-point to integer, or from integer to boolean. The latter means that tricks like the following are invalid:

```
int x;
if (x)   //In C/C++, means “if (x != 0)”.  Invalid in CFlat.
```

Such shortcuts are unreadable to many people since they don't make logical sense.

Note that in connection with the above, “if” and “while” statements now take a “bool” as a parameter, whereas they took an integer in C and C++; likewise, the relational operators (>, ==, etc.) return booleans, not integers.

The good news for programmers who like to use some of these implicit casts is that they can define them themselves as library functions. For example, defining a cast from int to bool is as simple as:

```
operator bool(int i) { return (i == 0)? false : true; }
```

Recall from section 4.2.5 that operator overloading is allowed between primitive types.

Note that explicit casting between fundamental types is still fine. So the following is valid:

```
int i := int(5.7);    //i gets the value 5.  “int i := 5.7” is invalid.
```

4.6.9. Case Else To Replace Default

In switch and select statements, the use of the keyword “default” is replaced by the phrase “case else”. The keyword “default” is not a keyword any longer, freeing it to be used as an identifier name. Indeed, it can be imagined that it is not rare for a programmer to want to create a variable named “default”.

5. CFlat 0.1 Compiler

As mentioned in the abstract, as part of my thesis I have written a compiler to implement a subset of the features described in section 4. I have tried to focus on implementing my new features only rather than features that are already in C++. The implemented features are listed in Table 1, which is in section 3. I call this subset of features CFlat 0.1.

For simplicity, the compiler translates CFlat code coming from standard in to C++ code on standard out, rather than dealing with files. The input and output files can then be specified using the redirection facilities of the command line program being used (usually "<" and ">").

In addition to the regular language features, I have added one just for this compiler that allows the inclusion of C++ code within CFlat code. This is done by enclosing code within "cpp {" and "}". So it would look like this:

```
cpp {
        //C++ code
}
```

This causes whatever is contained in the block to be copied verbatim to the output file, which is then compiled with a C++ compiler.

The compiler was written using lex for the lexical analyzer and yacc for the parser. It was tested with GNU flex and bison. Whereas lex and yacc normally expect C code within the actions that are executed when a rule is matched, I have used C++ code instead to make the compiler easier to write. This has worked fine, at least using flex and bison, because they copy the code within these actions verbatim. As long as I use a C++ compiler to compile the output from flex and bison, there have been no problems.

The compiler does not execute as fast as it could, nor was that my goal since it is mainly for testing purposes. For example, whereas a hash table is normally used for the symbol table handler, I have used the “map” template class in the C++ standard library. The drawback of this is that maps are sorted automatically, which is unnecessary for the symbol table. Optimization of the output code has likewise not been a priority.

Since the compiler is not meant to be complex, I have not yet dealt with variable scopes. This means that, for now, the same variable name cannot be declared twice within a program, regardless of where the declaration is placed in regards to curly braces ({ and }).

For ease of testing and debugging the compiler, I have also written two pieces of test code. One prints the token names of each token in the input (i.e. it stops after the lexical analysis phase). The other prints a “diagram” of the parse tree, showing the parse tree in a hierarchical fashion after it has been built but before the code generation step.

The CFlat compiler architecture is shown in Fig. 10.

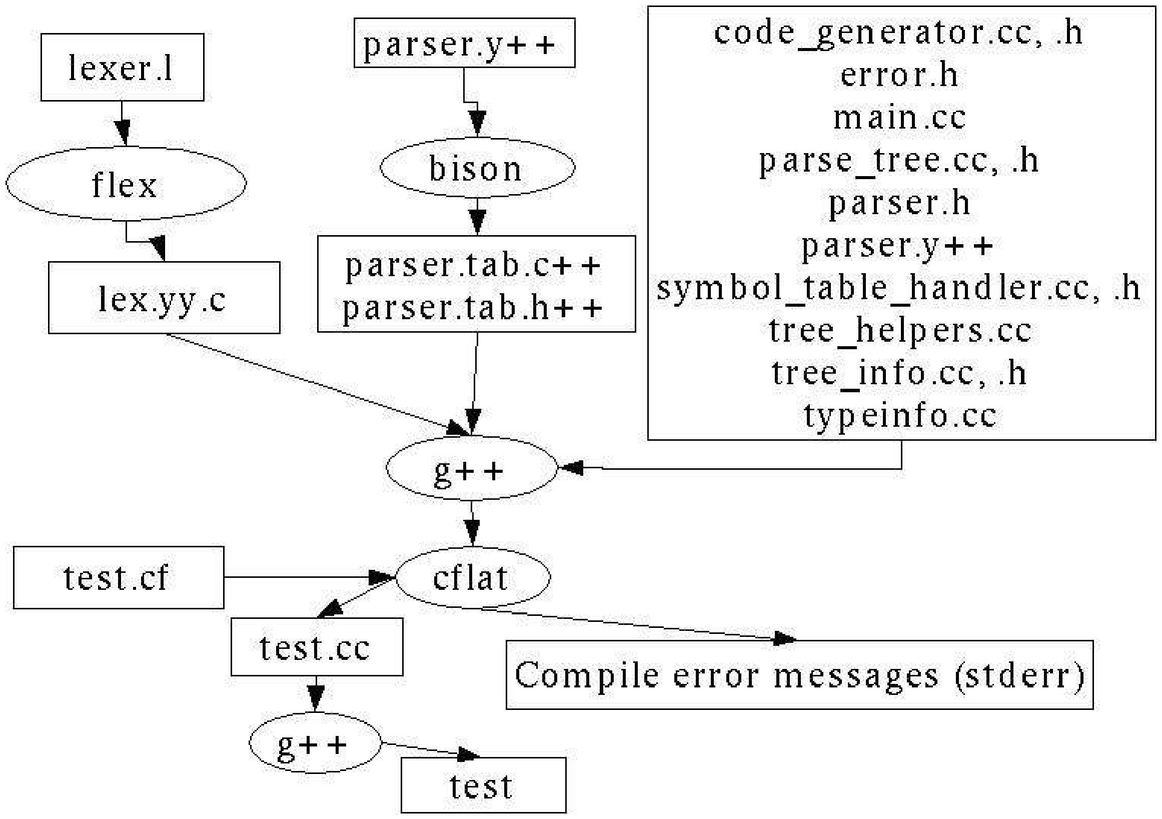

Fig. 10. CFlat 0.1 compiler architecture

6. Example Programs

The programs shown here are the ones used to test the CFlat 0.1 compiler. The last two are meant to contain compile errors and are for testing the error messages of the compiler.

```
/* Test program for CFlat.
Tests select, I/O, ranged case statements, inputting into a variable.
*/


<< "Enter a grade as a number: ";
>> int grade;

select (grade) {
          case 0: << "Really, really bad";
          case 1..49: << "F";
          case 50..59: << "D";
          case 60..69: << "C";
          case 70..79: << "B";
          case 80..89: << "A";
          case 90..99:
          case 100:  //test 2 case statements in a row
                    << "A+";
          case else: << "Invalid grade";
}
<< "\n";
```

```
/*Test program for CFlat.  Does computations on
the prices of products and the number of items
bought in a store.*/
double prices[-5..20];
prices[-5..-1] := 2.99;
prices[0] := 4.99;
prices[1..3] := 1.49;
prices[4..10] := 99.95;
prices[11..15] := prices[..-1];
prices[16..] := 0.0;  //reserved spots

/* Print all the prices*/
for (int i := prices.li..prices.ui) {
          << "Item " << i;
          select (prices[i]) {    /* select statement taking
                    a double (only ints allowed in C++) */
          case 0: << " is free.\n";
          case else: << " costs $" << prices[i] << ".\n";
          }
}

/*Array of bought items */
int bought[] :=    {-5, -2, 18, 2, 5, 7};
int numBought[] := {2,  1,  3,  9, 1, 0};
double priceSum := 0;

/* Print their prices in reverse order, and the final price */
for (int j := bought.ui..bought.li step -1) {
          << "Bought " << numBought[j] << " of item "
          << bought[j] << " for $" << prices[bought[j]] << " each.\n";
          priceSum := priceSum + prices[bought[j]] * numBought[j];
}
<< "Total spent: $" << priceSum << "\n";

prices[..] := 0.0;   //everything's sold
```

```
/*Test program for CFlat.
Calculates the sum and average of both the even and odd integers
from 1 to 100.*/
int[] Integers := {1..100};
int sum;

for (int start := Integers.li..Integers.li+1) {
          sum := 0;
          for (int x := start..Integers.ui step 2)
                    sum := sum + Integers[x];

          << "Sum of ";
          select (start) {
          case 0: << "odd";
          case 1: << "even";
          }
          << " numbers from 1 to 100: " << sum << "\n";
          << "Average: " << sum / (Integers.size \ 2) << "\n";
}
```

```
/* This file tests miscellaneous features that didn't get
tested in the other test programs */

for (int i := 10..1 step -1) << i;  //test negative step values

/*Test C++ blocks.  This shouldn't give a compile error once the
final result is compiled with a C++ compiler. */
cpp {
          class X {};
}

/* Testing nested comments
int x := 2;   /* Assign 2 to x */
<< x;     /* Nothing gets printed because we're in a comment */
*/

<< 3 - 2;   //Test subtraction
```

```
/* This file contains code that is supposed to produce
appropriate compile errors by the CFlat compiler.*/
int var := 5.2;   //Assigning a double to an int
int y;
y[1];  //Subscripting a non-array
double z := 5.2;
int x[10];
x[2..z];   //Using a non-integer in a range subscript
x[z..2];   //Test the left operand too
int z := 0;     //Duplicate definition
double p[];   //Declare an array without specifying a size
for (int i := 1..x) {}   //Use an array in a for loop range
for (int j[] := 1..10) {}  //Use an array as the loop counter
x[z];   //Subscript an array with a double
int u := x;   //Assign an array to a scalar
x[5.2] := 1;  //Assignment where subscript isn't an integer
x[{1, 2}] := 1;  //:= using an array (this isn't implemented yet)
x[5] := {1, 2};   //Assignment of array to scalar with subscripting
>> x;   //Input into an array isn't allowed
<< x;    //Output isn't either
{10..1} * 2;  //Multiply an array by a scalar (not allowed yet)
z.li;  //Can't access array properties for a scalar
z.ui;  //ditto
z.size; //ditto
x.property;  //Use a property other than li, ui and size
5.2 \ 2;   //Only integers can be divided with integer division
int v[10..8];   //Declare an array with upper bound < lower bound
int Integers := {1..100};  //Declare an array, forgetting the []
c;   //Use an undeclared variable
for (int k := 1..10 step 0);  //Step 0

/*Can't have an array in a select statement yet - requires an ==
operator, which must be programmed in CFlat rather than built in*/
select (x) {
case {1, 2}: {}   //Can't have an array in a case statement
case {3}..{5}: {}  //Ditto
}
```

```
/*This file tests errors that produce the generic "syntax error"
message, which causes compilation to stop.  For that reason it
must be separate from the other error checking program.*/
int z := 1;
int r[] := {z, 2, 1};   //Create an array using a variable
```

7. Conclusion And Future Goals

In my opinion, CFlat accomplishes my goal of improving on C++. I find it both easier to program in and more powerful.

It is difficult to say at this point whether the CFlat 0.1 compiler should be used as a base to write a complete compiler for the language. It may need to be changed to output an intermediate language other than C++ code, such as the Register Transfer Language (RTL) used by gcc [GCC Home Page, 2004]. It might also be easier to modify an existing C++ compiler to produce a CFlat compiler, due to the similarity of the languages.

The compiler still needs to become more efficient before it could be used for complex programs. It doesn't do any optimization, although existing optimizers could be incorporated into compilation, either by using the one in g++ or by optimizing the intermediate language, depending on the direction taking for the compiler's output language. At any rate, there is definitely a lot of work that needs to be done on the compiler to bring it on par with existing C++ compilers.

There are still other decisions to be made for the language specification itself. For example, I am as yet unsure whether garbage collection should be made a feature. I am considering including it but making it optional, though more investigation is needed. There are also many features I would like to include or consider including, but which wouldn't fit into this thesis report.

Of course, at this stage, the language is in a “beta” state; since there are not millions of lines of CFlat code in existence, any part of the language could change at any time. Ideally, before deciding for sure whether certain features are “good”, a compiler should be written and the features should be used to write large programs. I have not had the time to do that sufficiently for this project.

Overall, I am quite pleased with CFlat and the results of this work.

8. References

Bright, Walter, “The D Programming Language”, *Dr. Dobb's Journal*, vol 27, no 2, p 36, Feb 2002, further information available at http://www.digitalmars.com/d/.

GCC Home Page, RTL – GNU Compiler Collection (GCC) Internals, [Internet], [Updated 23 JUL 2004], [Cited 23 JUL 2004]. Available at http://gcc.gnu.org/onlinedocs/gccint/RTL.html.

Burgess, Mark (original author), *GNU C Programming Tutorial (Edition 4.1)*, [Internet], [Updated 23 SEP 2003], [Cited 30 JUN 2004], Available at http://www.crasseux.com/books/ctut.pdf.

Josuttis, Nicolai M., *The C++ Standard Library – a Tutorial and Reference*, Addison Wesley, Indianapolis, IN, 1999.

Kernighan, Brian, Ritchie, Dennis, *The C Programming Language Second Edition*, Prentice Hall Software Series, Prentice Hall PTS, Upper Saddle River, New Jersey, 1988.

Linden, Peter van der, *Just Java 2, Fourth Edition*, Sun Microsystems Press, Palo Alto, California, 1999.

Polivka, Raymond P., Pakin, Sandra, *APL: The Language and Its Usage*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

Sebesta, Robert W., *Concepts of Programming Languages Sixth Edition*, Addison Wesley, Colorado Springs, 2003.

Stroustrup, Bjarne, *The C++ Programming Language, Third Edition*, Addison Wesley, Indianapolis, IN, 1997.

Voegele, Jason, *Programming Language Comparison*, [Internet], [Updated 21 NOV 2003], [Cited 30 JUN 2004], Available at http://www.jvoegele.com/software/langcomp.html.

Appendix A. Features That Break C++

The following list of changes in CFlat from C++ cause C++ code to be invalid. The list is therefore useful in writing a program to convert C++ to CFlat.

- New keywords have been introduced: “function”, “returns”, “yield”, “yields”, and “select”. Identifiers with those names must be renamed.
- Programs may need to begin with “#include <lang>” since features that were built into C++ may now be defined there instead (section 4.1).
- Function headers have a new syntax (section 4.2.1).
- Overloaded operators have new syntax to match the function syntax (section 4.2.5).
- Array parameters to functions with [] will have to use * instead, since in CFlat there is a distinction between arrays and pointers (section 4.3.1).
- Where pointer arithmetic is done on arrays, it will have to be done on the address of the array instead; otherwise it will be considered an array operation (section 4.3.6).
- Since nested comments are allowed now, C++ block comments containing an unbalanced number of “/*” and “*/” pairs will have to be rewritten (section 4.6.2).
- Division of two integers with the “/” operator no longer does integer division; “\” must be used instead (section 4.6.3).
- floating-point constants with a decimal but no fractional part following the decimal (e.g. “4.”) are no longer allowed (section 4.6.4).
- The assignment operator has been changed from “=” to “:=” (section 4.6.5).
- The “->” operator is no longer available, and must be replaced with “.” (section 4.6.6)

- There are not as many primitive casts defined; these will have to be defined in a library and included if existing code using certain implicit casts is to continue to function (section 4.6.8).
- The “default” keyword is gone, and must be replaced with “case else” (section 4.6.9).

Appendix B. CFlat 0.1 Grammer

The below grammar is in EBNF. Keywords are in bold. Nonterminals are in italics.

```
program => statements
statements => statement statements
           | ε
statement => expr ;
           | declaration ;
           | for_header statement
           | { statements }
           | input_statement ;
           | output_statement ;
           | select ( expr ) statement
           | case_stmt statement
           | CPP_CODE
case_stmt => case expr :
           | case expr .. expr :
           | case else :
input_statement => input_statement single_input
           | single_input
single_input => >> new_or_old_variable
output_statement => output_statement single_output
           | single_output
single_output => << expr
           | << STRING
for_header => for ( new_or_old_variable := expr .. expr maybe_step )
maybe_step => step signed_intconst
           | ε
expr => assignment_expr
           | expr + expr
           | expr - expr
           | expr * expr
           | expr / expr
           | expr \ expr
           | ( expr )
           | INTCONST
           | FLOATCONST
           | variable
           | variable [ expr ]
           | array_constant
           | variable [ expr .. expr ]
           | variable [ .. expr ]
           | variable [ expr .. ]
           | variable [ .. ]
           | - expr
           | variable . IDENT
variable => IDENT
array_constant => { array_value_list }
array_value_list => array_value_list , array_list_item
           | array_list_item
array_list_item => FLOATCONST
           | signed_intconst
           | signed_intconst .. signed_intconst
assignment_expr => new_or_old_variable := expr
           | variable [ expr ] := expr
           | variable [ expr .. expr ] := expr
           | variable [ .. expr ] := expr
           | variable [ expr .. ] := expr
           | variable [ .. ] := expr
type_specifier => int | double
new_or_old_variable => declaration
           | variable
declaration => type_specifier IDENT array_info
           | type_specifier array_info IDENT
           | type_specifier IDENT
array_info => [ INTCONST ]
           | [ signed_intconst .. signed_intconst ]
           | [ ]
signed_intconst => - INTCONST
           | INTCONST
```

Appendix C. CS4997 Summary Sheet

PHASE TITLE	ESTIMATE		ACTUAL (approx.)	
	PERSON-HOURS	COMPLETION DATE	PERSON-HOURS	COMPLETION DATE
Grammar for CFlat 0.1	3	May 8	3.5	May 8
Lexical Analyzer	5	May 16	2.75	May 11
Write grammar in yacc	7	May 20	1	May 18
Symbol table handler	1	May 20	2	May 18
Reading	10	July 18	2.5	July 5
Yacc parser and code generator	30	June 3	59.5	July 31
Write test programs	2	May 27	5.2	July 23
Test and debug	20	June 13	15.5	July 31
Thesis outline	5	June 17	3	May 21
Thesis body	60	July 18	32	August 9
Prepare for seminar	20	July 31	11	August 6
Thesis revision	5	August 12	7.05	August 9
Total	166	August 12	145	August 9