CS 444 - Group 7
Design Documentation For Assignment 3

Bram Cymet, bcymet
Jung Myeng Lee, jm3lee
Thomas Kim, tkim

April 3, 2006

Contents

1 Design Changes From Assignment 2 **2**
1.1 Previous Design . . . 2
1.2 New Design . . . 2
1.3 New Type Hierarchy Organization . . . 3
1.4 New Type Resolution Algorithm . . . 4
1.5 New Design Versus Old Design . . . 4
1.6 Possible Improvements on the Design . . . 4
1.7 Symbol Table Changes . . . 4
1.7.1 The Algorithm . . . 4

2 Introduction **6**
2.1 Final List of Features . . . 6
2.2 Incompleted Features . . . 7

3 Runtime Environment **8**
3.1 Standard Library . . . 8
3.2 Exceptions . . . 9
3.3 Arrays . . . 10
3.3.1 Implementation Analysis . . . 11
3.4 Dynamic Range Checking for Non-Array Types . . . 11
3.5 Display Implementation . . . 12
3.5.1 `SymbolTable` and `SymbolTableEntry` Changes . . . 12
3.5.2 Implementation Analysis . . . 13

4 Code Generation **14**
4.1 The classes used in code generation . . . 14
4.2 Types of code generating algorithms . . . 14
4.3 Procedures, Functions, and Activation records . . . 14
4.4 Register Allocation scheme . . . 14

5 Testing and Debugging **16**

6 Generated Program Organization **18**

Chapter 1

Design Changes From Assignment 2

1.1 Previous Design

In assignment 2, an abstract syntax tree with the Visitor design pattern was used. The "VisitorStates" were used to modify the behaviour of the visitor on the fly. This seemed to provide a flexible way to hand-craft specific parts of the system. Please see Section 1.5 for the comparisons between the old and new designs.

1.2 New Design

To improve the funtionality of type checking a redesign was need so we consulted the wisdom of Richard M. Stallman. A pseudo-intermediate representation has been incorperated in the design. This data structure follows the list implementation described in [4] closley. Figure 1.1 depicts a typical organization for the input `a(a)`.

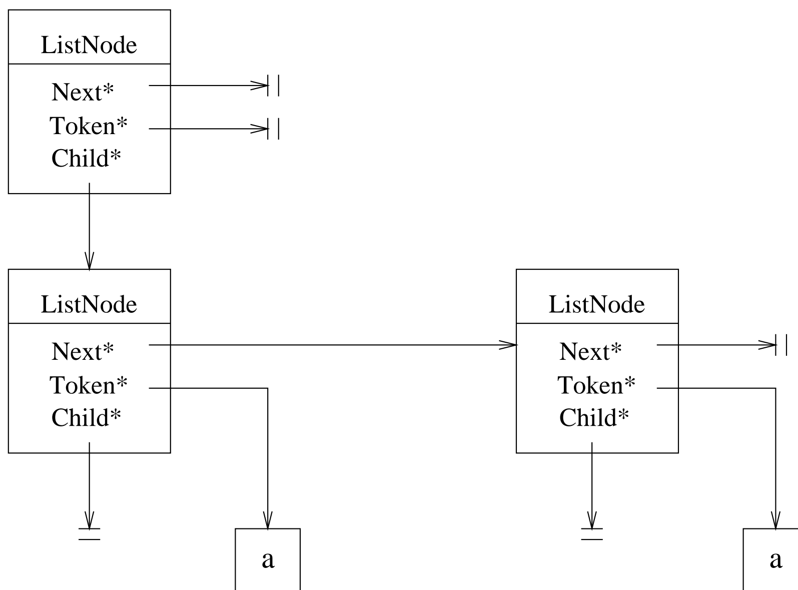

Figure 1.1: List Data Structure

A node may represent one of the following two things.

1. A list as in a functional language.

2. A symbol from source code.

With this data structure, it was possible to recursively traverse the list and perform type checking. This also simplified code generation as all information required was available in a condensed, uniform data structure. Note that this abstraction does not encapsulate the language constructs such as `if` statements. In our design, an `if` statement is abstracted as below.

```
if LIST then
    sequences of LIST
end if
```

In summary, this is a hybrid scheme with functional language like intermediate representation and an abstract syntax tree.

1.3 New Type Hierarchy Organization

The support for the keyword `new` was missing in Assignment 2. Due to this missing feature, we did not anticipate inputs such as

```
type Integer is range -10 .. 10;
subtype SubInteger is Integer range -5 .. 5;

type NewInteger is range -100 .. 100;
```

In the new design, we attempted to provide correct support for universal integers and floats. The `TypeDefinition` structure used in Assignment 2 has been extended to include a pointer to the parent type. For example, the above input would be represented using `TypeDefinition` structures as in Figure 1.2.

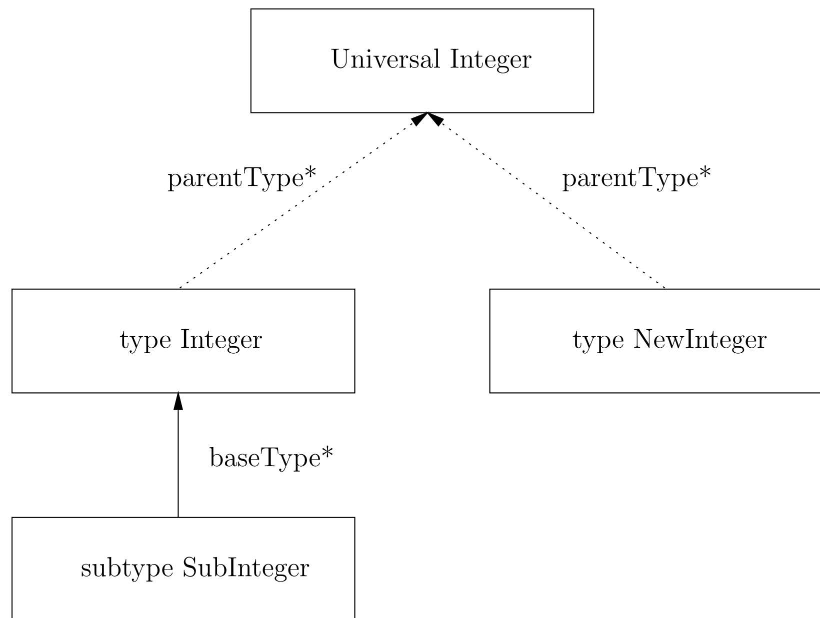

Figure 1.2: Type Hierarchy Example

The use of `parentType*` allowed us to implement a proper type checking system described in [3]. `baseType*` was used to resolve subtypes, whereas `parentType*` was used to resolve symbols so that a universal integer or float can be assigned to a variable of a particular type. Note that from our example,

`type Integer` and `type NewInteger` are considered as different types because there is no subtype relation between them. They only share the same parent type, which only classifies them as the siblings in the type hierarchy.

1.4 New Type Resolution Algorithm

Type resolution is performed in two phases. In the first pass, we resolve the tokens. In the second pass, we perform type checking. All these operations are performed recursively using the new data structure mentioned above. Although our `ListNode` resembles the implementation given in [4], we do not use λ-calculus to type check the input. Instead, we use our own algorithm to traverse the list.

1.5 New Design Versus Old Design

The "new and improved" design offered the following.

1. Localized type checking code. In the previous design, each visitor state had slightly different state from each other. This enforced us to specialize parts of type checking code for specific visitors. This lead to unneccessary duplication of code. Maintaining this type of code was very difficult.

2. A uniform representation of statements. An abstract syntax tree provided a recursive structure of input source code. However, each statement still has a different representation within the tree due to the differences in the context-free grammar rules that derive them. The new design unifies everything into one representation by treating every identifier and literal as a function. This greatly reduced the implementation complexity as the type checking algorithm for a function can be reused.

1.6 Possible Improvements on the Design

In retrospect, a complete intermediate representation in a functional language style would simplify the compiler even further. In our experience, the list abstraction has greatly improved the quality of the compiler. It also enhanced the maintainability. This further abstraction would let us

1. implement many language constructs such as loops and if statements as built-in functions. This would further reduce the source code size of the compiler.

2. achieve another level of enlightenment blessed by the functional language paradigm.

1.7 Symbol Table Changes

Our compiler did not support proper name resolution within nested scopes in Assignment 2. To correct this issue, a new entry has been added to the `SymbolTableEntry` class: `isShadowed`. Everytime an entry is added to a symbol table, other entries with the same names and definitions are shadowed. Shadowed entries are not eligible candidates in name resolution. The exact definition of "entries with the same definition" vary slightly depending on the type of an entry. Please see Table 1.1 for the exact equality testing criteria.

1.7.1 The Algorithm

On Insertion:

```
for each entries with same name
    if the new entry and the existing entry are the same
        shadow the exiting entry
```

On Deletion:

Entry Type	Criteria
Variable	Regardless of the variable type, if two variables have the same string representations, then they are equal.
Procedures and Arrays	The string representations have to match as well as the parameters.
Functions	Two functions are the same if they match according to the criteria for “Procedures and Arrays” as well as the return type.

Table 1.1: Criteria for `SymbolTableEntry` Equality Test

```
for each entries with same name
    if the new entry and the existing entry are the same
        unshadow the exiting entry
        break
```

Note that deletion of a symbol table entry happnes when scopes in a symbol table is decreased.

Chapter 2

Introduction

Assignment 3, the most difficult and time intensive part of the project, challenged our design and our will to code a lot. Due to the lack of time, we have implemented only a subset of features from Ada/CS.

2.1 Final List of Features

1. Ranged types
2. Array including dynamic checks for indices
3. Enumeration
4. Loops
5. If statement
6. Case statement
7. Exception
8. Subtypes
9. Non-local variable access
10. Exit statement
11. Subprogram
12. Parameter modes
13. Multiple variable declaration
14. Variable initialization
15. Partial support for strings
16. Overloaded functions including operator overloading
17. Write for Integer, Boolean and enumerated types
18. Read for Integer

2.2 Incompleted Features

1. Float
2. Records
3. Multiple packages
4. Dynamic range check for range types
5. Languaged defined attributes
6. Pragma
7. Use clause
8. Private types

Chapter 3

Runtime Environment

To support dynamic range checks in Ada/CS, we have implemented a library written in C++. At the code generation phase, the generated assembly code is linked with this library and generated range data. See Figure 6.1 for the overview of this process.

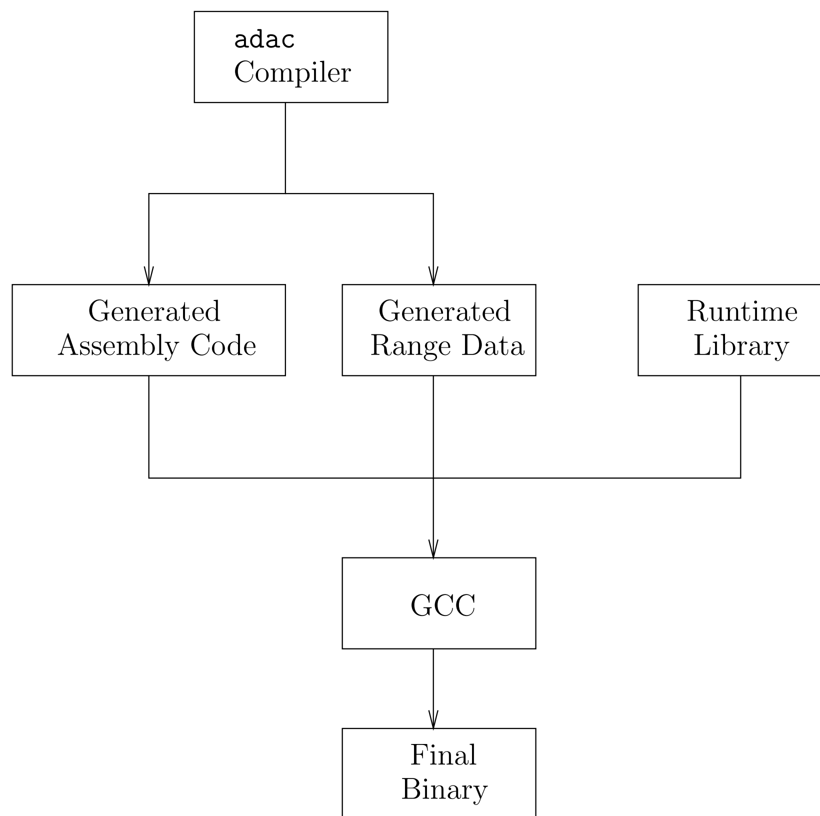

Figure 3.1: Runtime Environment and Linking Process

3.1 Standard Library

The standard library defines all the "primitive" types such as Integer, Float and Boolean. It also defines a few other subprogram listed in the Table 3.1. Note that most of these function could have been hard-coded

in assembly, but we decided to write the subprograms to demonstrate our compiler's functionality.

"mod" operator	"**" operator
"abs" operator	"and" operator
"or" operator	"not" operator
Overloaded WriteLine() with arguments String, Integer and Boolean.	

Table 3.1: Subprograms in Standard Library

3.2 Exceptions

The runtime environment also has the duties of managing the exceptions that are created along with the associated exception handlers. Refer to the diagram.

Whenever an exception is created, it is logged to the exception manager with the following information:

1. the frame it is local to, given by the local FP.
2. the exception value, given by the pointer address of the object used during compile time. This ensures that we have unique exceptions everytime we encounter them.
3. the address of where the exception should be handled.

Whenever we encounter a raise statement, the exception manager is called using the QueryException procedure. This will return the address of where this exception should be handled, if it is handled in this scope.

Every time we leave a frame, we must remove the exceptions that were local to that frame so that they wont be handled again.

The algorithm used to check exceptions and unwind the stack is given below:

```
check exception:
    check if there is an exception in the current frame
    if yes, remove the exceptions in the current frame goto handle exception
    if no, unwind the stack to the previous frame
    goto check exception

handle exception:
    remove the exceiptions in this frame
    execute the code for this exception
    return to the frame directly above this one
```

There were many challenges with this. To solve the problem with creating an exception but not defining a specific handler for it, we had to create a default handler for every frame. This allowed us to generalize the exception handling to handle any sort of exception.

So everytime a new frame is created, the exception manager will immediately log an exception for that new frame and the default handler.

With the exception manager and default exceptions, we were able to generalize the exception handling feature so that it would work in anycase.

Exception handlers are stored with two keys in the Runtime library: a frame pointer and an exception identification number. This leads to the design shown in Figure 3.2. Each map is implemented using the STL map class for the ease of implementation.

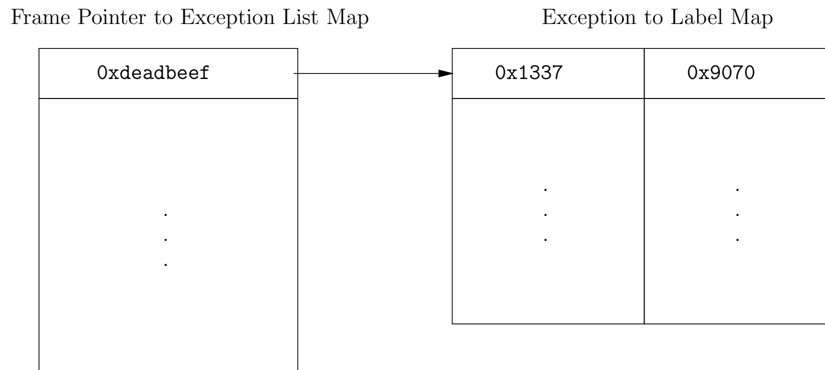

Figure 3.2: Exception Map Organization

We have listed the Runtime API for exceptions in the Table 3.2.

API	Description
`unsigned int QueryException(unsigned int fp, unsigned int exception)`	Query exceptions with the given keys: `fp` and `exception`. If the current frame does not have a handler for the given exception, return the base exception.
`void CreateHandler(unsigned int fp, unsigned int exception, unsigned int label)`	Insert exception handler denoted by `label` into the map with keys: `fp` and `exception`.
`void RemoveExceptions(unsigned int fp)`	Remove all the exception handlers with key `fp`.

Table 3.2: Exception Manager API

3.3 Arrays

For array implementation, we decided to implement a helper library routine to resolve the array index addresses, and perform dynamic array index range checking.

```
struct RangeInformation
{
    int             low;
    int             lowValid;
    int             high;
    int             highValid;
    int             offset;
};

struct Array
{
```

```
    unsigned int    dimension;
    void*           data;
};
```

The following shows an example of generated code for

```
type somearray is array(0 .. 10, 1 .. 5, a .. b) of integer;
```

where `a` and `b` are variables;

```
somearray:
    .long       0x3
    .long       some address

    .long       0x0
    .long       0x0
    .long       0x0

    .long       0x0
    .long       0xa
    .long       0x1

    .long       0x1
    .long       0x5
    .long       0x1

    .long       0x0
    .long       0x0
    .long       0x0
```

Note that the missing entries will be evaluated and populated at runtime.

3.3.1 Implementation Analysis

Advantages

1. It is easy to replace and extend the array management if we choose to do so.
2. The speed and ease of implementation was also a big factor.
3. We can uniformly handle arrays with single dimensions as well as multi-dimensions.

Disadvantages

1. The biggest disadvantage is that everything is on the heap. This leads to memory fragmentation as well as memory leak if one is not careful.
2. Performance suffers because of the lack of inlined array support. This means that creation, initialization and indexation suffers from library function calls.

3.4 Dynamic Range Checking for Non-Array Types

This feature has not been implemented. However, we provide preliminary design below.

Since the runtime library is written in C++, it is possible to take a full advantage of the `struct` data type provided by C++ in assembly. `adac` generates assembly code for each range defined in the source code as the following.

```
some_label :
                .long 0x0       ! Data type identifier (float or int)
                .long 0x0       ! Lower bound
                .long 0x0       ! Upper bound
```

This assembly code corresponds to

```
union RangeUnion
{
    int i;
    float f;
};

struct Range
{
    int dataType;
    union RangeUnion low;
    union RangeUnion high;
};
```

The "compiled" assembly source code would call one of the APIs provided by the library to check values against a particular range. It is clear that assembly programming can be minimized. This also helps us to debug the generated code in a systematic fashion. The runtime library provides the following API to perform the dynamic checking of ranges.

API	Description
`int CheckRange(struct Range*, RangeUnion value)`	This function checks if value is in the range described by `struct range`. If the value is invalid, it terminates the current program with `Constraint_Error`.

Table 3.3: Runtime Library API

3.5 Display Implementation

We closely followed the global display array implementation given in [1]. To avoid writing brain damaging amount of assembly code, we have extended our Runtime Library to include display management. A `struct` has been defined to represent the fixed portions of the frame.

```
struct Frame
{
    struct Frame* savedDisplay;
};
```

Table 3.4 lists the API provided in the Runtime library.

3.5.1 `SymbolTable` and `SymbolTableEntry` Changes

For the purpose of this discussion, we define a *logical scope* to be a block of code defined by a package, subprogram or block statement. Also, we define a *effective scope* to be a scope boundary at which a new frame is created. To clarify these definitions, please see the sample code below.

API	Description
`void InitializeDisplay(unsigned int number);`	Initialize display with the given number of entries.
`void InsertDisplay(unsigned int scope, struct Frame* frame)`	Insert the given address of a frame into the display list.
`void RemoveDisplay(unsigned int scope)`	Delete display entry at the given scope.
`unsigned int QueryDisplay(unsigned int i)`	Retrieve the ith display entry.

Table 3.4: Display Support in Runtime Library API

```
package foo is                  -- Logical scope 0, Effective scope 0
body
    procedure a is              -- Logical scope 1, Effective scope 1
    begin
        declare                 -- Logical scope 2, Effective scope 1
        body
            null;
        end;
    end;
begin
    null;
end;
```

Blindly relying on scope information maintained by the old code did not allow us to implement the display correctly. Our compiler in the previous assignment only maintained the logical scopes. However, to support correct type checking mechanisms, we introduced effective scopes into the system. Effective scopes let us clearly distinguish between local and non-local variables. Both `SymbolTable` and `SymbolTableEntry` classes had to be modified to support this feature.

3.5.2 Implementation Analysis

Advantages

1. The ease of development.

2. The ease of maintainance.

Disadvantages

1. This implementation is slower than an inlined implementation.

Chapter 4

Code Generation

Code generation was done with a mixure of using the Visitor Design pattern and the ListNode structure that was created for Type Checking.

Code generation using the ListNode structure is done in a very similar manner to type checking; it recursively traverses the nodes and creates the appropriate code.

4.1 The classes used in code generation

In order to faciliate code generation in a recursive manner, we created a class called the InstructionSequence to buffer the instructions being generated. The InstructionSequence basically keeps the instructions buffered in a list until it is ready for printing.

With this class, we are able to generate pieces of code that we can order in any way we would like. This helped us generate initialization code that we could place before any of the main body code.

Along with the InstructionSequence class, we created a RegisterManager class.

This class has the duty of managing all the available registers that can be used and a list of used registers.

For testing, we made it so that when we run out of registers it would assert false (do we really need to say this?). This indicated us that we were running out of registers. For basic operations, we should not be running out of any registers, so this testing method assured us we caught as many cases as possible.

4.2 Types of code generating algorithms

We had code generating algorithms for: Normal code used to evaluate expressions Address of code used for assignment statements Function generating code loops if statements case statements

4.3 Procedures, Functions, and Activation records

We chose not to use the register windowing feature for our code generation. The main reason was because we required total control over the layout of our activation record.

As a result, we had to design our own activation record. See 4.1

In, out, and in-out parameter modes were done according to the Ada spec. In variables cannot be modified, out variables are uninitialized and copied back, but in-out are copied to the callee parameter stack, and back.

4.4 Register Allocation scheme

Because we did not have enough time to implement a proper register allocation scheme, we implemented a very naive one. it basically uses registers only when it has to and then immediately store the values into

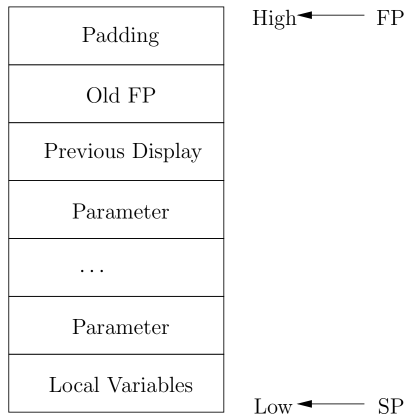

Figure 4.1: Activation Record

memory.

If we did have enough time, we would have implemented a graph colouring algorithm to optimally allocate registers as needed.

To facilitate this, we would simply make the register manager have an infinite number of registers, and remove the function where it stores the variable immediately to memory.

After we generate the assembly code, we would then pass it through our register allocation algorithm which would optimally allocate all registers. The main register allocation scheme that we would have used would have been from the [2].

Chapter 5

Testing and Debugging

Testing code generation was not as hard as type checking because we knew the expected out come of all our test cases, but debugging was very difficult.

We used gdb to step through assembly and to register dumps which would help us locate the errors.

Testing was done by writing small programs that would stress different parts of code generate.

For example, we made a matrix multiplication program to test multi-dimensional arrays. Also, with this test case, we modified it to test subtyping unconstrained arrays, new enumeration types, etc.

Also to test our procedure and function code, along with non-local variable access, we wrote heavily recursive and iterative test cases that stressed many parts that have to deal with procedures and functions. Please see Table 5.1 for the list of test cases.

Basic.adb	An empty program
BooleanExpr.adb	Boolean expression test
Call1.adb	Call an empty procedure
CaseTest.adb	Test case statement with enumeration values.
CountToInput.adb	Read subprogram
ExceptionTest.adb	Basic exception handling
ExceptionTest2.adb	Catching exceptions from a procedure
ExceptionTest3.adb	Catching a non-local exception
ExceptionTest4.adb	Catching an exception that not visible according to the scope rules
ExitTest.adb	Simple exit statement from a loop
ExitTest2.adb	Exit statement with a label
Exponent.adb	Prints the first 10 powers of 2
ExpressionTest2.adb	Stress test case for expression evaluation and register allocation
Factorial.adb	The classic recursive factorial calculation
Loop1.adb	An infinite loop using basic loop block
Loop3.adb	An infinite loop using while
LoopTest.adb	A comprehensive loop test case for all types of loops
ModuloTest.adb	Counts the number of integers that are divisible by 2 between 1 .. 100
StringAllocation.adb	Tests readonly data code generation
WriteTest.adb	A collective test for write subprograms
matrixArrayParameterPassingTest.adb	Multiply two matrices using subprograms
matrixSubtypeEnumerationTest.adb	Multiply two matrices that are subtypes of an unconstrained array with index type enumeration
matrixTest.adb	Basic matrix multiplication test
multiVariableInitialValueTest.adb	Variable initialization test
recursiveAdd.adb	Recursive summation of 10000 integers
recursiveAdd2.adb	The same as the above, but uses non-local variables
subtypeMatrixTest.adb	Multiply two matrices that are subtypes of an unconstrained array with index type Integer

Table 5.1: Test Cases

Chapter 6

Generated Program Organization

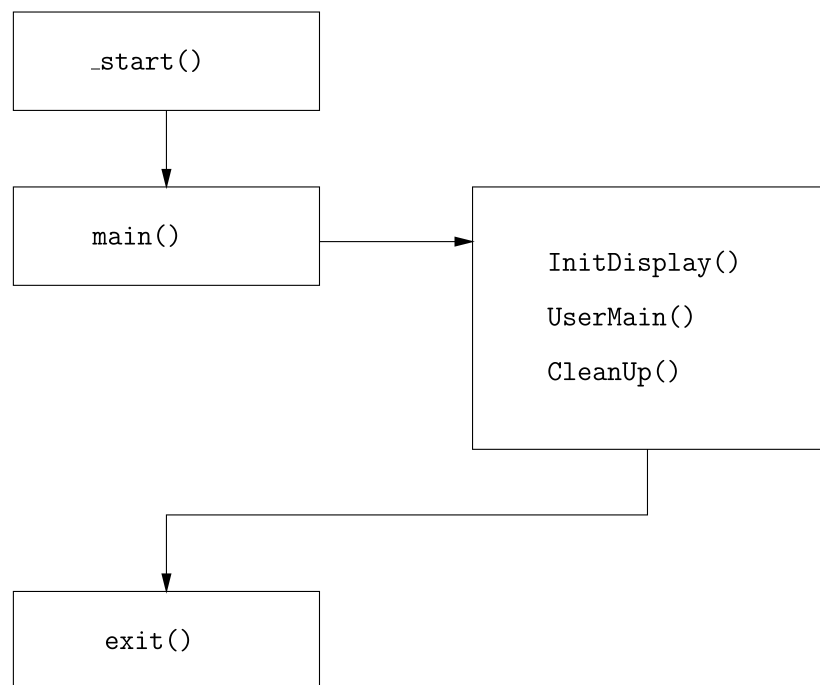

Figure 6.1: Overall User Program Flow

`_start()` is the entry point that `gcc` expects. `main()` also is the conventional name for the main entry point into the program. Our `main()` consists of three major operations.

1. Initialize the display data structure in the Runtime Library.
2. Run the user's main program.
3. Clean up internal data structures.

`exit()` triggers the proper termination of the program.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998.

[3] Kempe Software Capital Enterprises. Ada 95 reference manual. http://www.adahome.com/rm95/, 1997.

[4] Robert Krawitz, Bil Lewis, Richard M. Stallman Dan LaLiberte, and Chris Welty. *The Emacs Lisp Reference Manual*. Free Software Foundation, 2002.