Implementing the Dee System: Issues and Experiences

Lawrence A. Hegarty

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1992

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By : **Lawrence A. Hegarty**

Entitled : **Implementing the Dee System:**
Issues and Experiences

and submitted in partial fulfilment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee :

_____ Chair
Dr. D. Ford

_____ Examiner
Dr. J.W. Atwood

_____ Examiner
Dr. T. Radhakrishnan

_____ Supervisor
Dr. Peter Grogono

Approved by _____
Dr. C. Lam
Graduate Program Director

_____ 1992.

_____
Dr. M.N.S. Swamy
Dean of Faculty
(Engineering and Computer Science)

ABSTRACT

Implementing the Dee System:

Issues and Experiences

Lawrence A. Hegarty

The object oriented paradigm has been widely acclaimed as going a long way towards solving many problems addressed by the discipline of software engineering. Languages such as Eiffel, Smalltalk and C++ are examples of object oriented programming languages (OOPLs) that address these issues, but have not lived up to expectations. The Dee System is a pure, strongly typed object oriented programming language and an environment conducive to its use. Dee offers features not found in other OOPLs which enhance its ability to create robust, reusable and maintainable code. The implementation of the Dee compiler for Unix workstations is discussed. The methods used to implement several portions of the compiler are explained and suggestions are made about how to improve the implementation and the design of the Dee System.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Peter Grogono. His guidance and encouragement made my thesis work a pleasant and extremely educational experience.

I would also like to thank the "Dee team", WaiMing Wong, Joseph K.K. Yau, and most of all, my good friend Benjamin Y.C. Cheung. Never before have I had the opportunity to work with a better group of people. The success of the Dee Project and this thesis would not have been possible without them.

I would like to thank my girlfriend Viviane Guay. Without her confidence in my abilities, I would not have made it through this program.

More than anyone else, I would like to thank my grandparents, Mr. and Mrs. William Sankowsky. It was their continued support and encouragement that made this degree possible.

Contents

List of Figures **v**

1 Introduction **1**

1.1 Object Oriented Programming 1

1.1.1 Classes and Objects 2

1.1.2 Messages and Interfaces 3

1.1.3 Genericity 4

1.1.4 Exception Handling 6

1.2 The Dee System 7

1.2.1 The Dee Programming Language 8

1.2.2 The Compiler 10

1.2.3 The Browser 10

1.2.4 The Editor 11

1.2.5 The Linker 14

1.2.6 PC Dee 15

1.3 The Advantages of Dee 17

1.4 Motivation For The Dee Project 18

2 Implementation **20**

2.1 The Abstract Syntax Tree 20

2.2 The Scanner 22

2.2.1 The Generated Scanner 22

2.2.2 The Hand-Coded Scanner 24

2.2.3 The Two Scanners Compared 26

2.3 The Parser 27

2.3.1 The Bison Parser 28

2.3.2 Parser Evaluation 30

2.4 The Code Generator 32

2.4.1 The Run-time Data Structures 33

2.4.2 Control Structures 39

2.4.3 Applications 41

2.4.4 Assignments 45

2.4.5 Code Generator Options 46

2.5 Exceptions in Dee 47

2.5.1 Implementation 48

2.6 Dee Mode for Emacs 51

2.7 Garbage Collection 54

2.7.1 Types of Garbage Collectors 54

2.7.2 Generational Collectors 57

2.7.3 Garbage Collection in Dee 57

2.8 Base Classes 67

3 Conclusion **71**

3.1 The Dee System 71

3.2 Related Work 74

A The Abstract Syntax Tree **79**

B Class Int Implementation **85**

List of Figures

1.1 Eiffel code for class STACK 5

1.2 Dee System Organization 9

1.3 Sample Output from the ri command. 12

1.4 An example of the Dee mode browser under Emacs. The top window contains the program being edited. The cursor is on the name of the class being browsed in the lower window. 13

1.5 An X window running the Dfolder browser. 14

2.1 The Var portion of the AST data structure. 21

2.2 A Lex regular expression and its action. 23

2.3 Bison specification for Dee method bodies. 28

2.4 Loop portion of the AST. 30

2.5 Parser specification for PC Dee special methods. 31

2.6 Example of a Dee method 33

2.7 C code generated from the Dee method Entry for class Prime 34

2.8 Run-time Data Structures 35

2.9 The C declaration of a Dee object at run-time. 36

2.10 C code to calculate the size of an object. 37

2.11 C run-time data structure for a Class. 38

2.12 An example of an if statement in Dee. 40

2.13 The C code generated for the above Dee if statement. 40

2.14 A typical Dee loop statement. 41

2.15 The code generated for the above loop statement. 42

2.16 Basic form of an attempt statement. 47

2.17 Runtime class inheritance graph data structures. 49

2.18 The memory management structures used by the collector and allocator. 59

2.19 Code generator statistics. 67

2.20 The run-time data structure for an object. 68

2.21 The special Dee method to add two integers. 69

Chapter 1

Introduction

1.1 Object Oriented Programming

Object oriented concepts are not new to the field of computer programming. Almost all aspects of what we currently include under the umbrella of the object oriented paradigm were present in early programming languages. Work on SIMULA [**?**] began in Norway in the early 60's. The first version was a simulation language. The final version was designed to be a general purpose programming language. By the time SIMULA 67 was finished, it contained most of the features we commonly think of as being essential to an object oriented programming language. Other object oriented programming languages did not start appearing until about the early eighties, but they are becoming very popular.

Much of the motivation for the current popularity of object oriented programming languages is a direct result of the high cost of software creation and maintenance.

Maintenance can be described as modifying software that has already been delivered. These modifications can consist of error corrections or alterations as a result of design or specification changes. On page 536 in [**?**], Sommerville points out that "...maintenance costs are, by far, the greatest cost incurred in developing and using a system."

The rest of this section contains an introduction to some of the most important features of the object oriented paradigm. The final section of this chapter is an introduction to the Dee System.

1.1.1 Classes and Objects

A *module* is a syntactic unit of a programming language. Programs can be created by composing several modules, with each module having its own set of variables and procedures. The way in which other syntactic units of a program can access the internals of a module define its *interface*. A *class* is a kind of module; it is an abstraction of an entity that is manipulated by the software system. The difference between a class and a module lies in the type of interface they provide. In languages that support modules, a module can often export types, constants, variables and procedures. In most languages that support the object oriented paradigm, classes can only[1] export procedures that can be performed on instances of the particular class exporting them. In object oriented parlance, an instance of a class is called an

[1]Usually, a class also implicitly exports one type so that we can declare instances of the class.

object.

The class is the main conduit used in the object oriented paradigm to solve software cost problems. Unlike the techniques used in structured, top-down programming, system modularity using classes is more closely related to data than to functionality. When designing an object oriented system, one first looks at the objects that are manipulated by the system rather than the functionality of the system [**?**].

One example of an object manipulated by a system might be a stack. A particular system may have several stacks as data structures. Each of these stacks might be an object of the same class, that is, the class `Stack`. The ways in which an object of the class `Stack` performs stack operations is by receiving a request to act from another object.

1.1.2 Messages and Interfaces

A *method* is a procedure defined for a particular class. A *message* is a request by one object, to have another object execute one of its methods. This idea of exporting methods is related to the concept of a message. When a class exports a particular method it means that an instance of the class will accept a message that corresponds to a particular method. When a message is received by an object, it executes the appropriate method. Only messages that have been explicitly exported by the class may safely be received by instances of that class. It is the job of the type system and semantic analyzer to ensure that such unexpected messages are never sent.

Because in most object oriented programming languages data internal to an object

can not be exported, it can only be manipulated by methods exported by its class. In languages that do allow variables to be exported, this usually[2] makes them *read only* and saves the programmer from having to write a function that simply returns a variable's value. Exported methods may, in turn, invoke *private* (not exported) methods of the class, which may also alter an object's data. Still, in most cases, any change in the state of an object is ultimately the result of a received message, corresponding to a public method. This results in a high degree of data encapsulation and information hiding at the object level.

1.1.3 Genericity

Genericity permits the passing of type parameters to classes. It is similar to argument passing in functions. This passing of types allows one class to operate on many different types. In [**?**] Meyer uses the example of a stack. A portion of the Eiffel code for class *STACK* with a generic type parameter is shown in figure **??**

In this example, `T` is a formal generic parameter to the class *STACK*. Genericity allows us to use the same implementation of stack on several different data types. We can have a stack of integers, a stack of floats or even a stack of arrays of integers. Absence of genericity might force us to write a separate implementation for each data type. Untyped object oriented languages do not require genericity to achieve the same degree of code reuse, but do not reap the benefits of strong type checking.

[2]In some languages it is possible to make exported variables writable. The `friends` specifier in C++ is an example.

```
-- Stack elements of an arbitrary type T
class STACK[T] export nb_elements, empty, push, pop, top
feature

    push ( x : T ) is
      -- Add x on top of the stack
    do ... end

    top : T is
      -- the element on the top of the stack
    do ... end

    pop : T is
      -- remove the top element from the stack and return it
    do ... end

    .
    .
    .

end -- class STACK
```

Figure 1.1: Eiffel code for class STACK

1.1.4 Exception Handling

Exceptions are used to indicate unexpected or error conditions occurring in a section of code. Some examples of exceptions might be an attempt to divide by zero or to open a file that does not exist. Means to deal with these problems are built into several (not necessarily object oriented) programming languages including Eiffel [**?**], Ada [**?**], Smalltalk [**?**] and CLU [**?**]. A *signal* is the means used to indicate that an exception condition has occured. We use the term *raise* to describe the act of altering the flow of control with a signal. Usually, the method of dealing with an exception is to allow a non-local exit from the code in which it occurred. A *handler* is a section of code that has made known its willingness to deal with these singularities. When an exception occurs, a signal is raised and control is transferred to a handler.

An example used by Grogono in [**?**] is that of matrix inversion. It is difficult for a function to determine if a matrix is invertible without doing much of the calculation required to actually do the inversion. This makes it undesirable to check the precondition that the matrix is invertible before attempting the operation. Exceptions give us a way of neatly exiting the inversion function if it is unable to complete its task. In addition, an exception and handling mechanism built into the language gives the programmer greater flexibility in determining how and where a problem should be handled. This is accomplished by allowing exception handlers the option of dealing with the problem or passing it on to an outer scope of handlers.

1.2 The Dee System

Code reuse and efficient program maintenance are two important goals of almost every large software system. Although object oriented programming has frequently been offered as a way of solving these problems, the potential benefits of the object oriented paradigm have not yet been realized in currently available products. The Dee System is a complete software development environment created with the aim of solving these problems.

In designing the Dee system, the following six principles laid a foundation on which to build.

1. Information should not be duplicated.

2. All information related to a program entity should be in one place.

3. The programmer should not have to provide information that the compiler can easily infer and display.

4. Semantic analysis performed by the compiler should not be complicated.

5. A language that is intended for the development of production quality software should be strongly typed.

6. The language should support the chosen programming paradigm completely and consistently.

I present these principles so that the reader who is familiar with other object

oriented programming languages will understand some of the ways in which they differ from Dee. A deeper explanation of these design principles can be found in [**?**].

These principles were chosen because it is hoped that they enable the Dee system to satisfy its three primary goals.

1. a programming language that provides full support for the object oriented paradigm;

2. a programming environment for the language that supports all phases of software development; and

3. a library of classes that facilitate both coding at a high level of abstraction and efficient object code.

1.2.1 The Dee Programming Language

The focal point of the Dee Project is the Dee programming language created by Peter Grogono. It contains most of the trademark features of the object oriented paradigm including multiple inheritance, encapsulation, genericity, automatic garbage collection and exception handling. In addition it contains several features not previously found in programming languages. These additional features center on the data base of class interfaces maintained by the compiler and will be discussed below.

Figure **??** gives an overall picture of the Dee System and illustrates the way in which its individual components (rectangles) and data structures (ovals) interact.

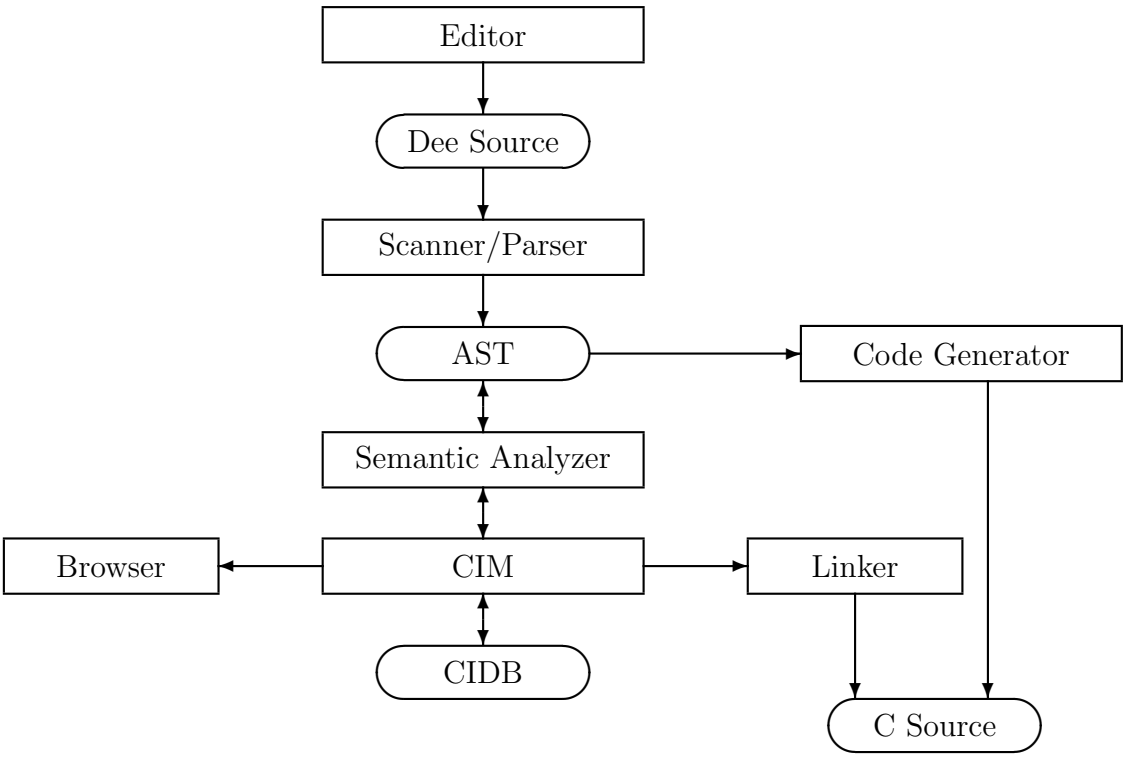

Figure 1.2: Dee System Organization

I have written the editor[3], the scanner, the parser and the code generator. I also created much of the Dee run-time system including the garbage collector and several base classes. In addition, I contributed a large amount of effort to the semantic analyzer originally written by Wai Ming Wong. I will now give a brief explanation of the most important parts of the Dee system. In depth explanations of the portions that I have contributed are presented in Chapter 2.

[3]Actually, I have created specialized editing features for the already existing editor called Emacs.

1.2.2 The Compiler

The Dee compiler can be broken down into several main components along the lines found in most compilers. There is a scanner, a parser, a semantic analyzer and a code generator. The scanner and parser create an Abstract Syntax Tree (AST) and the semantic analyzer decorates it. Finally, the code generator traverses it and outputs C code. Unlike other compilers, the Dee compiler has an additional component called the Class Interface Manager (CIM). This module acts as an interface between the compiler and the database of class interfaces. Whenever the semantic analyzer needs information about the interface of an ancestor or client class, it queries the CIM. If no errors are encountered during compilation, the interface to the class being compiled is also written to the CIM.

1.2.3 The Browser

The browser exists in several forms, the first completed and the most primitive of which is the `ri` command. The `ri` command was written to allow the programmers working on the Dee project to examine the interface to particular classes stored in the CIDB by making requests to the CIM. It was created to work as a test and verification tool rather than a programming aid. Only after the Dee compiler was first being tested did we realize it was quite useful to Dee programmers as a browser. It now serves as the basic tool with which more sophisticated browsers have been built.

The `ri` command outputs a description of the interface for the class specified on the command line. It has several options that determine the amount of information to display. Some of these options provide the ability to specify only one attribute, other options add information about how and where specific attributes are defined and implemented. Figure **??** is an example of a portion of the output of the `ri` command on the class `File`.

This primitive command has been combined with two more sophisticated programs to provide better browsing. The first is a Dee mode for the Emacs editor. When editing a Dee program, Dee mode allows a user to place the cursor on the name of a class and invoke the `ri` command on that class. A new Emacs window is opened with the output from `ri` displayed in it. Figure **??** shows an example of the browser being invoked during an editing session. See Section **??** for more details on Dee mode.

The second browser, written by Benjamin Cheung, runs under the X window system and is called "Dfolder". It is graphical, requires a mouse, and provides a friendly user interface. Options and classes can be selected by pointing with the mouse and clicking the mouse buttons. Figure **??** is a snapshot of the Dfolder running on an X window graphical display. See [**?**] for more detailed information on Dfolder.

1.2.4 The Editor

The Dee system has an editor at the center of its user interface. The compiler cooperates with the editor in displaying and fixing syntax and semantic errors. As

```
class File

inherit Stream
uses String Bool
Ancestors Output Input Device Stream

public var handler:Int
   source class: Device

public var option:String
   source class: Stream

public method close
   From
   Defined By : Device
   Implement By : Device

public method eof  : Bool
-- return ture if reach the end of file.

   Dee Instruction
   Defined By : File
   Implement By : File

public method open
-- Function open a file with its' option

   Dee Instruction
   Defined By : Device
   Implement By : File

public method read (n:Int ) : String
   Dee Instruction
   Defined By : Input
   Implement By : Input

public method write (buffer:String )
   Dee Instruction
   Defined By : Output
   Implement By : Output
```

Figure 1.3: Sample Output from the `ri` command.

Figure 1.4: An example of the Dee mode browser under Emacs. The top window contains the program being edited. The cursor is on the name of the class being browsed in the lower window.

Figure 1.5: An X window running the Dfolder browser.

errors are discovered, the editor brings the cursor to the delinquent line and allows the user to make a change before moving on to the next error. The editor also works closely with the browser to select which classes to browse.

A future version of the Dee editor will be graphical and probably run under the X window system. The current version uses the Emacs editor, which runs a special mode customized for Dee source programs. See Section **??** for more details.

1.2.5 The Linker

The Dee linker generates a link file from a collection of classes, starting from a given root class. This file is eventually linked into the final C linking phase which produces the Dee executable file. The linker may also generate a Unix "make" file which will

be used to control the C compiler and linker.

Linking in Dee is a two step process; there is a Dee linking stage and a C linking stage. This extra linking stage, not normally found in non-object oriented programming languages, is a result of dynamic binding. Because Dee allows inheritance, a variable at run-time can be an instance of one of many different classes. When a message is sent to such an object, the correct method must be invoked. The key to choosing the correct method is figuring out which is the correct slot in the method table. The compiler can not determine the correct slot because it is not until link time that the inheritance hierarchy becomes fixed. The same class, used in different programs, will have a different place in the run-time inheritance graph. The job of the Dee linker is to arrange these tables so that, in cases where more than one related class implements the same method, it is always at the same slot in the method table. The linker then assigns these offsets to variables used at run-time to index the tables. The linker uses table compression techniques in an effort to minimize the size of method tables. More information about the linker and its optimizations can be found in [**?**] and [**?**]. An explanation of how code is generated to allow dynamic binding at run-time can be found in Section **??** about applications.

1.2.6 PC Dee

Dee methods come in two flavors, regular and special. *Regular* methods are written in Dee. *Special* methods are code that can not be written in Dee, this has the added advantage of hiding implementation details. Special methods are actually written in

C. For example, the steps needed to open a disk file for output are dependent on particular operating systems. In our Unix implementation, the method `open` in the class `File` is a special method. It is written in C and uses the `open()` system call. Dee programs can avoid knowing about such system dependences because the class library hides these details.

The original implementation of Dee was written to run on a PC. This version of the language is different from the version currently implemented on Unix in several ways.

The biggest difference is in the back end of the compiler. The PC Dee version emits code for a virtual Dee machine while the Unix version emits C code. The Dee machine is simulated by a big switch statement with one label for each instruction in the virtual machine's instruction set. Eliminating the virtual machine in the Unix version altered the syntax for the *special* Dee methods. In the PC version, a *special* method was like any other Dee method except that it contained at least one virtual Dee machine instruction in its body. In the Unix version, because Dee machine instructions no longer make sense, a *special* Dee method consists of a normal method signature with its body replaced by the new keyword `special`. The Dee linker generates a standard C function prototype based on the current class and the name of the method. The programmer is responsible for writing the C function to match the prototype.

The Unix version of the language has been slightly altered to avoid using international characters. This resulted in replacing the use of the "[" and "]" characters by

"(" and ")" for specifying generic class parameters. Comments were previously surrounded by "{" and "}" and are now begun with two dashes ("– –") and terminated at the end of the line. This convention was borrowed from Ada because it has the advantage of never allowing run-on comments. That is, if the programmer forgets to include the comment terminating symbol (previously "}") the parser can incorrectly interpret a portion of the source code. This usually leads to a plethora of strange error messages being produced or a section of code not being compiled. With the Ada convention, it is almost impossible that the programmer will forget the end of line character.

Despite these differences, PC Dee is close enough to the new version to have been very helpful in making implementation decisions.

1.3 The Advantages of Dee

Dee is a true object oriented programming language. Thus Dee does not have the complexity of a hybrid language such as Ada or C++.

Dee is a strongly-typed language. Thus Dee programs are easy to read and do not fail at run-time. In this respect, Dee is more secure than untyped languages such as CLOS and Smalltalk.

Dee provides "semantic browsing." Semantic browsing depends on features of the language and cannot be added retroactively to other languages. The `short` and `flat` utilities of Eiffel [**?**] might at first sight seem to provide a capability similar to the

browsing features of Dee, but `short` and `flat` read the source text of a class. There is no guarantee that the compiled version of the class is up-to-date, or even that the class has been compiled. No information other than extracts from the source text is provided. The Smalltalk browser is similarly limited.

Dee provides fully automatic storage management. In Ada and C++, programmers are responsible for storage management. Errors in storage allocation and release are common and notoriously hard to find and correct [**?**]. Storage management adds to both the size and complexity of source code.

Class libraries are now available for Ada, C++, and Eiffel. All of them provide data *structures*. The class library of Dee is different in that it contains two levels of classes: data abstractions and data structures. Data abstractions specify *what* tasks can be performed and data structures describe *how* they are performed. Programmers code using data abstractions and only later, when the program is almost complete, choose appropriate data structures to represent the abstractions.

1.4 Motivation For The Dee Project

Reducing the high cost of software maintenance is one of primary design objectives of Dee. My work on the Dee project is ultimately concerned with determining how well this goal was attained. A fast and efficient compiler is only as useful as the language being compiled. An equally important result will be offering alternatives in places where Dee falls short of this goal.

My motivation for working on the Dee project comes from two different categories: issues involved in designing and programming in an object oriented language, and issues involved in writing an object oriented compiler and environment. Because the this is the first Dee compiler for Unix and because we have very limited resources, performance is not a primary goal. This is not to say that we paid no attention to this issue, just that reasonable performance was required, exceptional performance was not.

Despite the fact that I did not design Dee, I was present at many discussions which helped shape some of the language design decisions. Ultimately these decisions are judged on what happens when they are effected by the compiler writers and put to the test by end users writing applications.

Chapter 2

Implementation

2.1 The Abstract Syntax Tree

The heart of the Dee project is its Abstract Syntax Tree (AST). Many different parts of the Dee environment have an intimate relationship with the AST. The end result of the parsing phase of the compiler is an AST. The Class Interface Manager (CIM) then saves the parts of the AST, which represent the interface of the class, in the class interface data base (CIDB). The `ri` command, for browsing a class, uses the CIM to read a class interface in the form of an AST.

The definition of the AST used in the Unix version of the Dee system is not radically different from the one used in the PC Dee system. The first task I undertook when beginning the Unix system implementation was to convert the PC AST, written in Pascal, into C. Several fields were dropped, added, or changed slightly to reflect the new operating system and new Dee specification. The new C AST was the foundation

```
struct {
   StringPtr VarComment; /* Comment following variable */
   AST VarType; /* Type of the variable */
   Boolean VarPublic; /* True if public */
   AttriSrc AttributeSource; /* Where the variable is from */
   AST SourceClass; /* Added by the SA */
} Var;
```

Figure 2.1: The `Var` portion of the AST data structure.

on which all other parts of the Dee compiler were built. Having this accurate data structure at such an early stage was crucial to the success of our team of programmers. It allowed us to work separately on the different components of the system with only a minimum of communication. Wai Ming Wong was able to complete the alpha version of the semantic analyzer without having an AST to work with. After I completed the scanner/parser phase of the compiler, it became possible to test the semantic analyzer with real ASTs. The careful specification of the AST allowed us to put these two large components of the system together with virtually no changes (other than bug fixes). In a similar manner, I was able to complete a substantial portion of the code generator before I was ever able to test it with a decorated AST as produced by the semantic analyzer.

An example of cooperation and communication between different components of the Dee system via the AST can be seen in the example of the `Var` section of the AST presented in Figure **??**. The complete AST definition can be found in Appendix A.

Different fields of this variant of the AST structure are filled in by different parts the compiler. The parser fills in the `VarComment`, the `VarType` and the `VarPublic`

fields. This information comes directly from the source of the class being compiled. The semantic analyzer then interacts with the CIM to fill in the last two fields. If there are no compilation errors, the parts of the AST relevant to the class interface are written to the CIDB. Finally, the entire decorated AST is passed to the code generator where it is traversed one last time to produce code.

2.2 The Scanner

Scanning is the process of dividing the input stream into groups of characters called tokens. I have created two different scanners for Dee. The original scanner was generated by a scanner generator. Because lexical analysis is considered to be one of the most time consuming parts of the compilation process, a new hand-coded scanner was written. This second scanner is different in several ways. It resulted in a smaller compiler executable, but did not achieve a significant speedup.

2.2.1 The Generated Scanner

The original scanner for Dee was generated by the standard Unix utility *Lex* [**?**]. It has the advantage of being easy to write and modify. The specification of the scanner used by Lex is only about 100 lines long. The scanner it creates is compatible with parsers generated by *Yacc*.

Lex takes as input a set of regular expressions [**?**, **?**]. It then generates tables that are used by a finite automaton to recognize tokens. When each regular expression is

```
[A-Za-z_]([A-Za-z0-9_])*
              {
                 int t;
                 t = Lookup(yytext);
                 if ( t == IDEN ) {
                    yylval.IdenVal = Hash(yytext);
                    yylloc.last_line = TokenLine;
                    return t;
                 }
              }
```

Figure 2.2: A Lex regular expression and its action.

matched, a corresponding segment of user specified C code is executed. This segment of code does things like add symbols to the hash table, record comments and string literals, and return the token type.

The portion of the Lex specification for the Dee scanner in Figure **??** recognizes identifiers and keywords. The left column is a regular expression describing a string of characters starting with a letter or an underscore and followed by any number of characters which may be a letter, number or underscore. Whenever this regular expression is matched, the C code on the right is executed. Since this one regular expression describes both keywords and identifiers, the C code checks a table to differentiate the two. If a keyword is found, its token value is returned to the parser. If an identifier is found, it is hashed and the token for an identifier is returned to the parser.

Using a finite automaton is particularly well suited to scanner generators because the code that interprets the tables does not depend on the lexical structure of the

language to be scanned. It is only the tables that change for each new set of regular expressions. According to [Waite], this method of lexical analysis is not as efficient as embedding the translation logic into a program.

Flex is a Lex compatible scanner generator which is slightly more efficient than the original. The specification file used by Lex can be used, unaltered, by Flex. Benchmarks used to compare generated and hand-coded scanners were run on both Lex and Flex.

2.2.2 The Hand-Coded Scanner

Both [**?**] and [**?**] make several suggestions for writing a hand-coded scanner which I incorporated into the design of the new Dee scanner. The most important of these are the use of a large input buffer and the indexing of a table based on the current input character's ASCII value. Using large buffers for source input was carried to an extreme in my Dee scanner.

The most efficient way to read the source text into main memory is to avoid reading amounts which would only partially fill a buffer. Because disk controllers, the operating system and even the routines that do the high-level reading may contain buffering, it is difficult to know the optimal amount to read. This problem has been minimized in the hand-coded scanner by reading the entire source file with one call to the operating system. This allows each level of buffering to perform optimally with the exception of the last buffer of the file, which will probably be partially full. In addition, this significantly improves processing speed by eliminating the need for

checking when the end of a buffer is reached. Now we need only check for the end of the file.

Traditionally, entire source files were not read into main memory because their size might exceed the compiler's (or the operating system's) ability to hold them. Classes tend to be small enough to easily fit into the very large address spaces found on modern hardware. The logical size of a source file in Dee, and in most object oriented programming languages, is one class. The data in table 2 demonstrates the trend of these source files, to be rather small. The numbers in the table are the sizes of files measured in bytes. Because only a limited number of Dee classes have been written, I also included data for classes of other object oriented programming languages. As the number of Dee classes rises, we can also expect the average class size to increase. This increase should bring the Dee averages closer to those of the other languages cited.

Language	Avg. in bytes	Max. in bytes
Dee	890	2822
Eiffel	2370	5821
C++	9746	18850

The hand-coded scanner was also written to be compatible with parsers generated by Yacc.

2.2.3 The Two Scanners Compared

In order to compare the speed of the two types of scanner I wrote a test driver. It does nothing but repeatedly call the scanner until the end of file is reached. I then created a shell script to run the scanner, on a large class, 200 times. This shell and the Unix `time` command combined to produce the average times listed in the table below. The time column indicates the total real-time the command took to execute, measured in seconds. The size column indicates the size of the executable in bytes.

Scanner	Size in bytes	Avg. time
`lex`	68779	2.1
`flex`	68815	2.0
Hand-coded	65728	1.9

Variations in the load on the network resulted in a margin of error. Because the results are so close we can not conclude that any one scanner is significantly faster than the others. Still, we can argue that with speed not a factor, machine generated scanners are preferable.

The difference in speed between the three different scanners is not large enough to justify using it as the sole metric on which to judge the scanners. Any change in the language that requires a change in the scanner will be significantly easier to effect using Lex or Flex. This is particularly important in young languages such as Dee.

Changing the machine generated scanners mostly involves altering the regular

expressions, with the exception of the small amount of C code executed on a token match. When the scanner generator is re-run, it will use the new regular expressions to generate new tables. Because the tables in a hand coded scanner are maintained by the programmer, altering them is a tedious and difficult task.

The hand-coded scanner does have several advantages over a machine generated scanner. It can determine the exact position of the token on a line. Knowing the column a token starts in can facilitate better error reporting. Both Lex and Flex can determine the line on which tokens are found, but are unable to keep track of which column the token begins or ends. The hand-coded scanner results in a smaller executable as noted in the table. It is possible that a language may require tokens to be matched which are not expressible as regular expressions[1]. In this case, a hand-coded scanner is required.

2.3 The Parser

The job of the parser is to determine if a given source program is syntactically correct. If syntax errors are found, they should be reported to the user in a clear and concise manner. If no errors are found, the parser is responsible for organizing the source program in a form suitable for semantic analysis and, eventually, code generation. This form, in the case of the Dee compiler, is an Abstract Syntax Tree (AST). The Dee parser is generated by *Bison*, a parser generator from GNU. It is upward compatible

[1] $\{\ [^i]^i \mid i \geq 1\ \}$ is a well known example. Such scanners cannot handle nested comments.

```
MethBody      : BEGIN StatementList END
                                { ThisBody = BodyConcrete;
                                  $$ = $2; }
              | SPECIAL
                                { ThisBody = BodyInstr;
                                  ClassHasSpecial = TRUE;
                                  $$ = MakeInstr(); }
              | FROM IDEN
                                { ThisBody = BodyFrom;
                                  $$ = MakeIdent($2); }
              | /* empty */
                                { ThisBody = BodyAbs;
                                  $$ = NULL; }
              ;
```

Figure 2.3: Bison specification for Dee method bodies.

with the standard Unix parser generater called Yacc.

2.3.1 The Bison Parser

The Bison parser generator accepts as input, a specification for the language to be parsed. This specification is in the form of a grammar augmented with a set of actions. The grammar must be context free and expressible as an LALR grammar. The specification is basically a machine readable grammar in Backus-Naur Form. The actions are a list of C statements that are executed when their corresponding syntactic rule has been recognized. Figure **??** is an example of a rule and its action from the Bison specification for Dee.

This rule indicates that a method body (`MethBody`) can come in one of four forms. It can be a list of statements surrounded by the keywords `begin` and `end`, the

keyword `special`, the keyword `from` followed by an identifier, or it can be empty. `StatementList` is a rule itself and is defined elsewhere in the specification. The tokens in all capital letters are terminal symbols (i.e. `SPECIAL`, `FROM`, `IDEN`, and `BEGIN`). A terminal symbol is recognized by the scanner and is not comprised of constituent parts recognized by the parser. Comments come in the same form as those found in C (i.e., `/* empty */`).

In order to compile a program, the compiler must know more than simply an integer has been parsed, it must also know the value of that integer. Bison has a special semantic value mechanism for each token. Each terminal and nonterminal type can have a semantic value. For terminal symbols, this often comes in the form of an integer value, float value or identifier name. For non-terminals it usually comes in the form of a structure comprised of several semantic values.

The action part of the rule is written as a list of C statements. Special symbols embedded in the code can be used to signify the semantic value of each token. The parser generator replaces these special markers with their corresponding semantic values. In the first part of the example in Figure **??**, the `$$` stands for the semantic value of `MethBody` and the `$2` stands for the semantic value of the second constituent of the rule being recognized. In this example, it stands for the semantic value returned by `StatementList`. The semantic value of `StatementList` is returned as the semantic value of `MethBody` if the first portion of the rule is recognized. Another example can be seen in the action associated with the `from` clause. In this case, the `$2` is used to pass the semantic value returned by `IDEN`.

```
struct {
   AST FromStmts;    /* List of init. statements */
   AST UntilCond;    /* Boolean expression */
   AST WhileCond;    /* Boolean expression */
   AST LoopStmts;    /* List of loop statements */
} Loop;
```

Figure 2.4: Loop portion of the AST.

The action taken when some non-terminals are matched results in the creation of an AST. This tree is a simplified version of the source file. It is simplified in the sense that many of the reserved words are not present in the tree. Their meaning in the language is retained by the structure of the particular parts of the tree that they correspond to. Take, for instance, the part of the AST that represents a Dee loop in Figure **??**.

We are only interested in the list of statements that make up the `from` part of the loop. If there were no statements, `FromStmts` will point to nil. Similar protocols are followed by the other parts of the loop representation.

2.3.2 Parser Evaluation

It is not uncommon for academic language compilers to make use of tools such as Bison and for commercial production compilers to use hand coded parsers. Commercial compilers tend to exist only for languages that have gained some degree of popularity. With this extensive use, a language gains maturity and is less like to have changes in its syntax. Academic languages, on the other hand, are often susceptible

```
MethBody        :       INSTR
                                    { ThisBody = BodyInstr;
                                      ClassHasSpecial = TRUE;
                                      $$ = MakeInstr( $1 ); }
                ;
```

Figure 2.5: Parser specification for PC Dee special methods.

to modifications. The choice of which parser to use is primarily governed by two criteria: efficiency and ease of modification. For established languages, the tradeoff in modifiability is worth the gain in speed. For academic languages such as Dee, the loss in efficiency is a small price to pay for the ease with which a machine generated parser can be modified.

One example of the evolution of Dee syntax involves the way one writes *special*[2] methods. The original version of the specification can be seen in Figure **??**.

`INSTR` was recognized by the scanner as an identifier beginning with the character '!' (ie. !add). This notation came from the PC version of Dee. The funny looking identifier actually corresponded to an instruction in the virtual Dee machine on which PC versions of Dee programs run. On the Unix version of Dee, special methods are implemented directly in C. This means that the name of the method and the class in which it is found is enough information to make a special method unique. The funny identifier was replaced by the keyword `special`. Because I used a machine generated parser, this change was easy to effect. I simply added the new keyword to the hash

[2]Special methods are the way in which Dee interfaces with other languages and with the operating system.

table of keywords and changed the parser rule to look like what was seen in Figure ??.

Hand coded parsers are more complicated from the compiler writer's point of view and are much more prone to errors. The simplicity that makes machine generated parsers so easy to alter also has the added benefit of making them less likely to contain errors. The Dee parser has been relatively free from bugs since the very beginning of the development cycle.

While speed is not sufficient cause to write a hand coded scanner, there is one flaw in Bison generated parsers that may warrant this outlay of labor. The error reporting capabilities are inadequate in machine generated parsers in general and this is especially true in the case of Bison. It is quite easy to get the parser to report that a syntax error occurred on a particular line. It is quite difficult to get it to report a more meaningful message and to recover in a reasonable way. Hand coded parsers are better able to give sophisticated error messages and recover from errors.

2.4 The Code Generator

The *Code Generator* is the section of the compiler that takes a decorated AST as input and produces C code as output. The heart of this section is the routine `Gen()`, which recursivly traverses the AST. Each method in the AST that is implemented in the class being compiled (with the exception of `special` methods) causes one C function to be generated.

```
method entry
-- Generate all the prime numbers up to
-- max using Eratosthenes' sieve
   var f:Filter i:Int max:Int
   begin
      max := 2000
      f.make(2)
      from i := 3
      until i > max
      do
         f.process(i)
         i := i + 1
      od
   end
```

Figure 2.6: Example of a Dee method

Figure ?? is an example of a typical Dee method. It is the constructor for the class `Primes`. Figure ?? contains the C code generated for the Dee method in Figure ??. The name of the C function corresponding to a Dee method is created by prefixing the name of the Dee method with the name of the class it is implemented in and an underscore.

2.4.1 The Run-time Data Structures

Dee methods compiled into C functions do not use the C stack to pass Dee arguments or return values. Instead, all parameters and results are conveyed using the Dee stack. The *Dee stack* is a fixed size array of pointers to objects. This stack is used in much the same way the C stack is used. It contains a stack frame for each currently active method.

```
void Primes_entry( osp )
   int osp;
{
   int Old_hwm = hwm;
   os[osp] = NIL;
   os[osp+2] = NIL;
   os[osp+3] = NIL;
   os[osp+4] = NIL;
   hwm = osp+5;

   os[osp+4] = create_int( 2000, hwm );
   os[osp+2] = NIL;
   os[osp+2] = CreateInstance( _ClassTableIndex_Filter, osp+2 );
   os[osp+6] = os[osp+2];
   os[osp+7] = create_int( 2, osp+7 );
   Call(Filter_M_make,5);
   os[osp+3] = create_int( 3, hwm );
   do {
      os[osp+6] = os[osp+2];
      os[osp+7] = os[osp+3];
      Call(Filter_M_process,5);
      os[osp+6] = os[osp+3];
      os[osp+7] = create_int( 1, osp+7 );
      Call(Int_M__plus,5);
      os[osp+3] = os[osp+5];
      os[osp+6] = os[osp+3];
      os[osp+7] = os[osp+4];
      Call(Int_M__gt,5);
   } while ( os[osp+5] != true_object );

   hwm = Old_hwm;
} /* entry */
```

Figure 2.7: C code generated from the Dee method `Entry` for class `Prime`

```
struct Object {
   ClassPtr Class;
   struct GCFields Status;
   union {
      int Int;
      ObjectPtr Bool;
      char *String;
      double Float;
      unsigned char Byte;
      ArrayPtr ArrayBody[1];
      ObjectPtr InstVars[1];
      int GCStuff[2];
   } Tag;
};
```

Figure 2.8: The C declaration of a Dee object at run-time.

Figure **??** is an example of the run-time data structures, including a portion of the Dee stack. The first pointer in each stack frame is a pointer to the return value of the method. It exists, but is ignored, in methods that do not return a value. Following the return value is a pointer to the current object. This is the object that contains the method we are executing and whose instance variables we have direct access to.

Figure **??** shows the C declaration for the run-time structure of an object. The first field contains a pointer to the class descriptor of which the object is an instance (see below for a more detailed description of a class descriptor). The second field is used only for garbage collection and is explained in Section **??**. The last field is a union which is used differently depending on which class the object is an instance of. The first five fields in the union are used if the object is a base class. Even in a pure object oriented language, the basic building blocks of more complicated objects

```
AddOn = ( NumInstVars - 2 > 0 ) ?  NumInstVars :  0;
CorrectObjectSize =
   sizeof( struct Object ) + AddOn * sizeof( * struct Object );
```

Figure 2.9: C code to calculate the size of an object.

must eventually have a representation in a form directly usable by the CPU. Even though Array is not a base class, it is a special case. The field `ArrayBody` is a pointer to a structure that contains information specific to arrays. This includes the bounds of the array and the actual array of pointers to objects. The field `InstVars` is an array of pointers to objects. For objects that are not of class `Array` or a base class, this array is sized to the number of its instance variables. C can be *tricked* into creating variable sized structures that allow us to declare `InstVars` to be only length two, but actually be the correct size for objects that have more than two instance variables. The C code in Figure **??** calculates the number of bytes needed for an object which has `NumInstVars` instance variables. The last field in the object data structure, `GCStuff`, is used only when the object is reclaimed by the garbage collector and placed on the free list of objects. It would have been possible to avoid this extra declaration, but it helps keep the garbage collection code simple by avoiding the need for frequent use of casts. Since the `GCStuff` field is in a union, it costs nothing to include. See Section **??** for more details.

Each class used, whether through inheritance or a client relationship, by a particular Dee program contains a corresponding class descriptor at run-time. The C declaration for a class descriptor contained in Figure **??** is also of variable size in

```
struct Class {
   ClassType ClassKind;
   int InstanceSize;
   int InstanceCount;
   Parent ParentArray;
   MethodPtr *Methods;
};
```

Figure 2.10: C run-time data structure for a Class.

much the same way the object structure is. In the case of the Class structure, it is the number of methods that determines the size of the structure. The first field, `ClassKind`, is a scalar that specifies whether the class is a base, array, or regular class. The `InstanceSize` field is the size of an instance in bytes. `InstanceCount` is the number of instance variables in each instance of the class. `ParentArray` is an array of pointers to the class structures of all parents of the class. It is used for testing conformance at run-time and is explained in more detail in Section **??**.

Referring back to Figure **??**, we see that above the pointer to `self`, are the arguments to the method. Not every method has arguments, but if it does, they are placed above `self` on the Dee stack, not on the C stack. Finally, the top-most portion of the stack frame contains the local variables for the method. Upon entry to the method all locals are initialized to point to the special object `NIL`. The field `result` is also set to `NIL` before the method is entered. The `NIL` object contains as many methods as the class with the highest index into its method table. Each method pointer for the class `NIL` points to the *illegal method*. This method prints the message "`Attempt to send message to uninitialized object.`" and aborts

the program. Setting uninitialized objects to point to `NIL` prevents Dee programs from crashing without an explanation if a message is sent to an uninitialized object. The last field, `Methods`, is the variable array of pointers to functions used to dispatch methods at runtime. This field is covered in more detail in the Section ?? about generating code for method calls.

Two C variables are used to keep track of locations on the object stack. `Osp` is the object stack pointer. It always points to the base of the current stack frame. When a C function corresponding to a method is called, its only argument is the new value of `osp`. This is calculated in the calling method by adding the number of parameters and locals variables in the current stack frame to the current object stack pointer. The global variable `hwm` (for high water mark) is always set to the highest location in use on the object stack. It is used by the garbage collector to scan the object stack for all live objects. Any locations on the stack above `hwm` contain garbage. See Section ?? for more details.

2.4.2 Control Structures

The mapping of Dee control structures to C control structures is quite simple. `If` statements in Dee are directly mapped to `if` statements in C augmented with gotos. The gotos are needed because checking the Dee conditions may require some C code to be executed. An example of a simple Dee `if` statement and its corresponding C code is found in Figures ?? and ??.

Dee `loop` statements are mapped into C `while` statements. Notice that the

```
if i < 5 then
   k := i + 1;
elsif i > 10 then
   i := 0
else
   b := true;
fi
```

Figure 2.11: An example of an `if` statement in Dee.

```
os[osp+6] = os[osp+2];
os[osp+7] = create_int( 5, osp+7 );
Call(Int_M__lt,5);
if ( os[osp+5] == true_object ) {
   os[osp+6] = os[osp+2];
   os[osp+7] = create_int( 1, osp+7 );
   Call(Int_M__plus,5);
   os[osp+4] = os[osp+5];
   goto IF1;
}
os[osp+6] = os[osp+2];
os[osp+7] = create_int( 10, osp+7 );
Call(Int_M__gt,5);
if ( os[osp+5] == true_object ) {
   os[osp+2] = create_int( 0, hwm );
   goto IF1;
}
os[osp+3] = true_object;
IF1:;
```

Figure 2.12: The C code generated for the above Dee `if` statement.

```
from
   i := 0
while
   i < 5
do
   i := i + 1
od
```

Figure 2.13: A typical Dee `loop` statement.

method `Int_M_lt`, used to do the comparison `i < 5`, is called twice. The first call is needed because the condition must be checked before the body is executed. If the condition fails, the body will not be executed at all. If the body is executed at least once, the condition must be calculated again. Calling `Int_M_lt` the second time leaves the truth value of the condition on the object stack for the C `while` statement to recheck. The only difference in the code generated for Dee `while` and `until` loop statements is that the condition is negated in the `until` version. Figure **??** shows the Dee code for a simple loop and Figure **??** shows its corresponding generated C code.

See Section **??** for information on the control structures used in `attempt` statements.

2.4.3 Applications

An application is the sending of a message in object oriented parlance. In more traditional terms, it is the invoking of a procedure. The first step in generating code for an application is determining if the method is a constructor. This can be

```
os[osp+2] = create_int( 0, hwm );
os[osp+6] = os[osp+2];
os[osp+7] = create_int( 5, osp+7 );
Call(Int_M__lt,5);
while ( os[osp+5] == true_object ) {
   os[osp+6] = os[osp+2];
   os[osp+7] = create_int( 1, osp+7 );
   Call(Int_M__plus,5);
   os[osp+2] = os[osp+5];
   os[osp+6] = os[osp+2];
   os[osp+7] = create_int( 5, osp+7 );
   Call(Int_M__lt,5);
   } /* while */
```

Figure 2.14: The code generated for the above `loop` statement.

determined by examining information placed in the AST when it was decorated by the semantic analyzer. If it is a constructor, the receiving object must first be created. This is accomplished by calling the C function `CreateInstance()` with an argument specifying the class of the object to be created.

After the type of the method is determined, the next step is to set up a new stack frame above the current stack frame (see Figure **??** for an example of a stack frame at run-time). The new result field is set to `NIL`. A pointer to the object receiving the message is copied to the self field. All the arguments are calculated and copied into the argument slots above the self slot. Calculation of the arguments often results in additional messages being sent and, thus, additional stack frames being prepared above the frame currently being built. When additional stack frames are used to calculate arguments, they are set up so that they will leave their result in the slot for the argument. After all the arguments are calculated and stored in their correct

slots, the method is ready to be invoked.

The following is an example of the C code to call the method `make` in class `Filter`.

```
os[osp+off+1] -> Class -> Methods[Filter_M_make](osp+off)
```

The variable `off` points to the beginning of the new stack frame. `Os[osp+off+1]` is the slot in the object stack that points to the object receiving the message. The field `Class` points to the class structure of this receiving object. The field `Methods` is the table of pointers to C functions corresponding to the methods for the class which self is an object of. The variable `Filter_M_make` is an index into this table, which was generated by the Dee linker. By making this a variable, we are able to insure that the correct method is dispatched as long as the receiver is of the correct type.

For example: Let there be a variable `X` of class `Shape` and an application of the method `show` to `X`.

```
var X : Shape
⋮
X.show
```

In addition, there are three classes `Circle, Square` and `Oval`, which all inherit class `Shape`. `Square` and `Oval` both implement their own versions of the method `show`.

```
class Circle
inherits Shape
```

```
⋮


class Square
inherits Shape
method show
   begin
   ⋮
   end
⋮


class Oval
inherits Shape
method show
   begin
   ⋮
   end
⋮
```

At run-time, it is possible that `X` will be an instance of class `Shape, Circle, Square` or `Oval`. But no matter what class `X` happens to be a instance of, for this particular application, the method `show` for that class will be invoked. The Dee linker guarantees that the pointer to the method `show` is at the same offset in the method table for

each of the classes that inherit `Shape`. If the method `show` is implemented in class `Shape`, it too will be at that same offset. Since that offset can only be determined at link-time, that Dee linker emits a declaration for the integer variable `Filter_M_make`, and assigns the correct offset to it.

2.4.4 Assignments

Assignments come in two forms: assignments to local variables of the currently executing method and assignments to the instance variables of the object currently executing a method. It is illegal to assign to an instance variable of any object but the current object. Dee syntax rules prevent these types of assignments.

In the following example,

```
foo := 3
```

if `foo` is a local variable, the C function `create_int()` is called to create an instance of the class `Int` with a value of 3. This object is then assigned to the correct slot in the active stack frame. If the offset for the variable `foo` is 4, the code generator would produce the following C code for this example.

```
os[osp+4] = create_int( 3 );
```

If `foo` is an instance variable it must belong to the object in the self slot of the currently active stack frame. Because instance variables may be inherited in much the same way methods are inherited, we index the table of instance variables with

a variable that is defined by the Dee linker. The code generated in this second case would be as follows.

```
os[osp+1] -> Tag.InstVars[SomeClass_V_foo] = create_int( 3 );
```

We know which slot in the stack frame points to the current object because *self* is always in the slot one above the top of the stack. The variable `SomeClass_V_foo` is generated in exactly the same way as was the variable used to access method table in the previous example except that a `_V_` replaces the `_M_` used in method table index variables.

2.4.5 Code Generator Options

There is one command line argument to the Dee compiler, which is passed to the code generator. By placing a `-V#` after the name of the file be compiled, where `#` can be 0, 1, 2 or 3, the user can control the verbosity of the C code generated. Zero is the default and produces C code which is not particularly easy for humans to read. As the numeric portion of the `-V` switch gets closer to 3, the code generator emits increasingly more verbose C code. The code produced by using an argument of 3 is not legal C code and can not be compiled, but is especially easy for humans to understand. For level 3, local variable stack offsets are replaced with the Dee name of the local variable. This feature is particularly useful when attempting the difficult task of debugging the generated code or when trying to write Dee `special` methods.

```
attempt
   S
handle x_1 :  C_1
   S_1
⋮
handle x_n :  C_n
   S_n
end
```

Figure 2.15: Basic form of an `attempt` statement.

2.5 Exceptions in Dee

Dee has facilities for generating and handling exceptions (see Section **??** for an explanation of these terms). An exception in Dee comes in the form of an exception object. For example, the following line of code raises an exception of class *String*:

```
signal "Division by zero."
```

An exception object can be an instance of any class. When the exception is signaled, the exception object is bound to the *exception register*.

The programmer can write code to handle the exception by using the `attempt` statement whose generic form is shown in Figure **??**.

If an exception is generated[3] by S, the system will attempt to determine if it matches any of the handlers in the current `attempt` statement. This is accomplished by testing if the exception object conforms to each class C_n starting with n equal to 1. If the exception object does conform to a class C_n, it is bound to the corresponding

[3]And it is not caught by a handler at level closer to the code that generated it.

variable x_n and the statements, S_n, corresponding to that handler are executed. If no match is found in the current scope of handlers, the next level of exception handlers is tried. Eventually, if no match if found, the run-time system will print the message "Unhandled exception." and cease execution.

2.5.1 Implementation

When an exception occurs at run-time, the system will have to determine which handler will accept it. Because of the semantics of Dee, it is necessary to test the exception object for conformance to the class of each potential handler. This test requires that the inheritance graph be available at runtime.

The inheritance graph is actually embedded in the class structures that exist at runtime. Each class structure has a field (`Parent`) that points to a variable length array of integers terminated by a -1 in the last slot. If the class has no parents, then `Parent` points to `NULL`. The integers contained in the array are indices into the global array (`ClassTable`) of pointers to the actual class structures.

It would be quicker to test conformance if `Parent` were an array of pointers to the classes. This would avoid an extra indirection but the addresses in memory of the class structures are not known at link-time. It would be possible to generate code that, when the program is first run, would fill in `Parent` array, but this would slow down the start up time for all programs.

The C function `Conforms()` is called at run-time to determine if the class of the exception object conforms to the class of each handler. If a match is found,

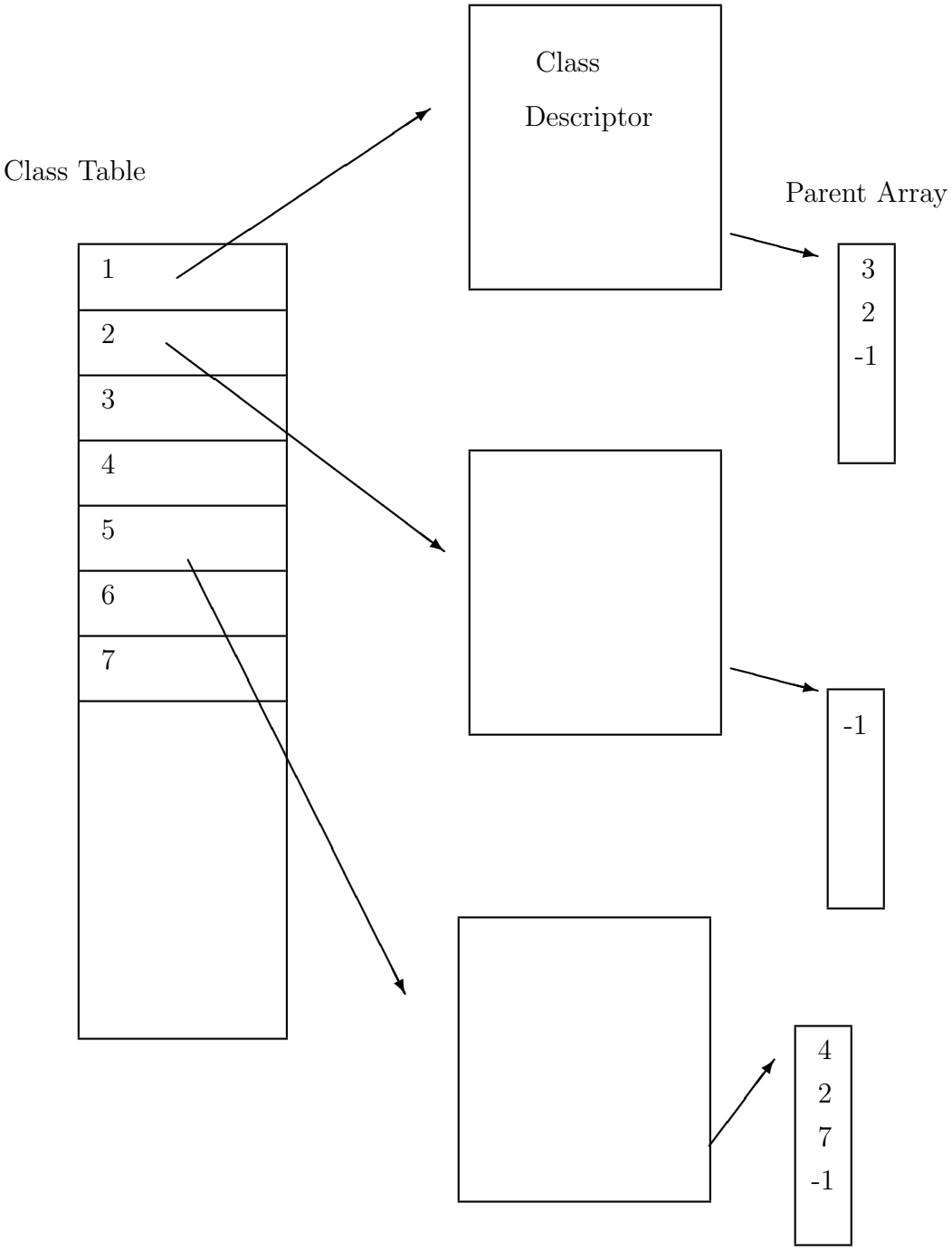

Figure 2.16: Runtime class inheritance graph data structures.

the exception object is bound to the identifier in the signature for that handler and the code for that handler is executed. `Conforms()` tests conformance by recursively traversing the list of parents, contained in the `Parent` array, all the way up the inheritance graph trying to find an exact match between the class of an ancestor and the class of the exception object. If such a match is found, `Conforms()` returns true, otherwise it fails and the next potential handler uses `Conforms()` again. Because inheritance sub-trees tend not to have an extremely large number of nodes (classes), traversal is not an expensive operation to perform.

The C functions `setjmp()` and `longjmp()` are used to save and restore context. Each time an `attempt` statement is entered, a new block of C code is created with a local variable to save the current context using `setjmp()`. After the context is saved in the local variable, it is placed on a stack of saved contexts. When a signal is raised, we do a `longjmp()` to the context at the top of the stack. If no handler is found, we pop the next context on the stack and do a `longjmp()` to its stored context. Declaring each element on the list as a local variable does not require us to allocate and deallocate memory each time an `attempt` statement is entered. If the statements inside the the `attempt` body complete without causing an exception to be raised, the stack is popped but no `longjmp()` is executed. The action of popping the stack removes the element from the stack. The variable is reclaimed automatically when the block is exited. This technique is a modified version of the ideas presented in [**?**] and [**?**].

2.6 Dee Mode for Emacs

Emacs is one of the most commonly used editors used on Unix systems. Much of this popularity is due to its power and flexibility. Emacs is partly written in C and partly in Emacs Lisp. The C part of Emacs consists of a Lisp engine and a large number of functions callable from Lisp. A significant part of the editor is written in Emacs Lisp and runs on this engine. To extend Emacs one writes additional Emacs Lisp code or changes the existing code.

When editing a program using Emacs, the editor is usually in a *major mode* which provides editing features specific to the type of file being edited. Major modes exist for the C and Lisp programming languages and for natural languages.

I have written a major mode for Dee, to assist programmers in writing Dee source code. It can insert the skeleton of all the major programming constructs. For instance, typing the key sequence `C-c l` (a control-C followed by the letter 'l') will insert the following text at the cursor

```
from

while

do

od
```

and indent it correctly. It will leave the cursor after the `from` statement as this is

where the user will most likely start typing. Having the editor insert text avoids mistakes in syntax and spelling that are usually caught by the compiler. This, in turn, speeds up the development cycle.

Whenever a tab or return is hit, Dee mode will automatically attempt to correctly reindent the current line and, in the case of a return, indent correctly on the new blank line. Aside from saving the programmer many keystrokes, this indentation can aid in the early detection of syntax errors. It is usually easy for the programmer to know how much any given line of code should be indented. If Emacs does not indent a line as the programmer expected, it usually indicates a mistake in the code above that line.

Because Dee mode prevents and detects errors at an early stage in the process of writing programs, it reduces the amount of time the programmer spends in the edit-compile loop. This, in turn, increases programmer productivity.

Perhaps the most useful feature of Dee mode is its built-in browser. When the cursor is placed on a word that is the name of a class already written to the CIDB (Class Interface Data Base), the interface of that class can be viewed. After invoking the Dee mode browse-class command with the keystrokes `C-c b`, an additional window is opened and the interface for the selected class is displayed in it. Every time a new class is selected with these keystrokes, its interface is shown in this window.

In the case of a class interface that is frequently referred to, the user can browse in a different way. By selecting the class in the same way, but by then invoking the perm-browse-class command with the keystrokes `C-c B`, a permanent window is

created and the interface for the selected class is displayed in it. This window is permanent in the sense that browsing another class with the browse-class command will not overwrite it. As many of these permanent browse windows can be created as needed.

If a class that has already been permanently browsed, is selected again, the old interface is overwritten. This is because the class may have been recompiled and the system must be careful never to present two different interfaces to the same class at the same time.

The browser in Dee mode is not meant to replace a more user friendly X windows browser, but it does have its place. Although Emacs can be used with a mouse, it was primarily designed to work without one. Experienced Emacs users often prefer to avoid touching the mouse whenever possible. Incorporating the browser in Dee mode allows them this freedom. Many users of Dee will not have X windows or other suitable windowing environment, they may have insufficient hardware or be working over a phone line. Dee mode gives them all the functionality of a graphical browser but without the fancy (and probably more friendly) user interface.

Emacs may not be the primary editor in the complete version of the Dee system, but it is a valuable alternative. Many Unix programmers already know how to use Emacs and should not be forced to use a special Dee editor. Experienced Emacs users have the option of performing their own customizations in addition to those provided by Dee mode.

2.7 Garbage Collection

Garbage collection is the process of reclaiming memory that is no longer needed by a program. Memory can safely be considered *garbage* when it is no longer reachable from global variables, active stack variables or through a chain of pointers starting from these locations. When a portion of memory is *collected* it is returned to the pool of memory available for future use. Because Dee is a pure object oriented programming language, memory that the garbage collector deals with is always in the form of an object. I will therefore use the term *object* to describe the unit of memory that garbage collectors operate on. In other languages, memory might contain smaller units such as integers, pointers or characters.

2.7.1 Types of Garbage Collectors

The most common types of garbage collection include reference counting, mark-and-sweep, and stop-and-copy. This section contains a full explanation of these algorithms. The first collectors where implemented by McCarthy [**?**] and Collins [**?**] in the early 60s. These were of the mark-and-sweep and reference counting variety. As technology increased, virtual memory allowed large heap space. As a result, stop-and-copy collectors started appearing in the last 60s.

In addition to the methods mentioned above, Goldberg and others [**?**, **?**, **?**] have shown that strongly typed programming languages can implement garbage collection which is specific to each program. This method has the advantage of avoiding tags,

used to determine the type of an object, normally found in conventional collectors.

By having the run-time system automatically reclaim memory when it becomes garbage, we free the programmer from this difficult task. Boehm and Weiser [**?**] show that when garbage collection is added to C programs, previously unknown memory leaks are often found.

Reference Counting

Reference counting [**?**] basically consists of keeping a count of the number of references to each object. When the count drops to zero, the object is unreachable and thus garbage. It may then be collected. Reference counting has several disadvantages. It is not possible to reclaim a circular list of objects even though it is not reachable from the active variables. Because the list contains a loop, none of its components have a zero reference count. Ways to overcome this problem have been implemented, but they often have a negative affect on the programming style because programmers are required to make consious efforts to avoid loops. This restriction reduces the main benefit of automatic garbage collection, that it should be transparent to the programmer. Reference counting requires space for a reference count and maintaining this count consumes considerable overhead. In some implementations, every store instruction has some garbage collection overhead associated with it.

Mark-and-Sweep

Mark-and-Sweep collection is a two phase process. The *mark* phase is a traversal of

all live memory. A bit is set in a tag associated with each portion of memory that is reachable. The *sweep* phase then reclaims all memory that does not have the bit in its tag set. This algorithm has the disadvantage of needing to traverse the entire address space of a process in order to return unmarked objects to the free store. It also has a space overhead of at least one bit per object. Unlike reference counting, mark-and-sweep algorithms do reclaim unreachable circular structures.

Stop-and-Copy

Stop-and-Copy collectors avoid the overhead of traversing all of a process' memory by copying all live data to a new area. Once all live data in a region has been copied out, the entire region can be reclaimed without being examined in detail. To copy the live data out of a region the algorithm traverses only the live data. As each live object is reached, it is copied to a new region. After the copying is completed, all pointers must be updated to reflect the new location of each object. This technique requires more memory than other techniques because memory must be divided into (at least) two different regions, each large enough to hold all live objects with room left over. Stop-and-copy collectors have an advantage in that they do not scan all memory, only that which contains live objects. In addition, copying collectors compact data as it is copied to the new region, eliminating the need for a separate compaction phase.

2.7.2 Generational Collectors

Generations are an extension to the above collection algorithms that allow objects

that have been alive for longer periods of time to avoid being considered for collection. Generational collectors were first introduced by Liberman and Hewitt [**?**]. They make use of the fact that as objects age, they are less likely to become garbage. In this type of collector, heap space is divided into several sections called *generations*. Each time the collector is invoked, it increments the age counts of all live objects. When an object reaches a threshold age it is copied to the next generation. This process of moving objects to a new generation is called *promoting*.

As a result of promoting, objects that are less likely to become garbage are concentrated in the older generations. When space is needed, the collector can affect only the youngest generations where the ratio of garbage to active objects will be highest. Collection in older generations occurs less frequently than in younger generations, usually when not enough space was reclaimed from passes on earlier generations.

Generations have been added to both mark-and-sweep collectors [**?**] and stop-and-copy [**?**].

2.7.3 Garbage Collection in Dee

The Dee garbage collector is of the mark-and-sweep variety. To avoid some of the disadvantages normally associated with this type of collector, it was augmented with generations. Currently, the Dee collector implements four generations but this can be easily changed. Each generation consists of a linked list of large pieces of memory called *chunks*. Chunks are obtained through calls to *malloc()*. Dee Objects are allocated from these spaces and are never split across chunk boundaries. See Figure

Figure 2.17: The memory management structures used by the collector and allocator.

?? for an illustration of these data structures. The design constrains Dee objects to be no larger than the size of a chunk. Chunk size, currently 4K, is hard-coded into the run-time system but may become a compile time option in a later version of the Dee system.

Free space is maintained by keeping a linked list of unused areas pointed to by a header unique to each chunk. The nodes in the list are portions of unused space right in the chunk. Each node in the list contains a field indicating how much space is free at that point, and pointers to the next and previous free slots. Keeping a doubly linked list will allow us to change the algorithm used to maintain the free list without much difficulty. Currently, free space is returned to the front of the list and is allocated by a first fit algorithm.

Whenever a new object is needed, the routine `Oalloc()` is called instead of

`malloc()`. `Oalloc()` traverses the list of chunks in the first generation until it finds one with a slot large enough to satisfy the request. The slot is removed from the free list and its address is returned. If the slot was bigger than the amount requested, the remainder of the slot is kept on the free list unless it is less than the minimum size of an object (this special case is discussed below). If no acceptable slot is found then a new chunk is allocated and the new object slot is taken from it. Before the new chunk is allocated, the garbage collector is invoked. Even if enough space is recovered to satisfy the current call to `Oalloc()`'s needs, a new chunk is still allocated. This helps prevent the collector from being called too frequently as a result of only a small amount of memory being reclaimed.

Fragmentation and padding

One of the weaknesses of the current Dee implementation is that the minimum size of an object is 16 bytes on most 32-bit processors. This is a result of design decisions made before garbage collection was considered, but it does result in some benefits for the collector. If a slot being allocated from the free list is bigger than the new object but not so big that the remainder could contain another object, it is treated as a special case. Instead of being placed on the free list, this extra memory is allocated with the memory requested. Eight bits in each object are used to determine how much *padding* each object has. This has the disadvantage of wasting bits in each object and bytes in fragmented chunks. Padding objects has the advantage of making compaction unnecessary. Whenever an object is reclaimed, it is always at least large

enough to contain a base object. In practice, not many bytes are consumed by fragmentation because base objects (which are always the minimum legal size of an object) are created and reclaimed so frequently. Slots smaller than the minimum object size can never exist on the free list. When a chunk is swept, adjacent garbage slots are compacted together further eliminating fragmentation.

Marking

During the mark phase of collection, each active object must be flagged in some way. After the sweep phase, these flags must be cleared. The method used by Zorn [**?**] maintains a bit map for each chunk of memory. The mark/test/clear operations on a bit map have the advantage of being very localized but expend the overhead of a table lookup. Locating the tag next to, or in, the object with which it is associated results in touching each active object three times for each collection: once to set the bit, a second time to test it when sweeping and a third time to clear all the set bits in preparation for the next collection. Localization becomes a factor on the clear operation. If each active object must be touched a third time, we lose much of the advantage gained by having generations. To solve this problem and avoid the expense of table lookup associated with the bit map method, I have used a mark field of eight bits instead of one. This allows me to set the mark field to the current mark counter. When sweeping, I simply test the mark field with the current mark counter; a match means the object is alive. The mark count is incremented once in each collection phase, but zeroed when it reaches 255. There is no need to clear the mark field as

long as I ensure that no object ever goes more than 256 collection cycles without being swept.

Sweeping

Sweeping occurs on a per generation basis. After a mark phase, any or all of the generations may be swept depending on the current configuration of the collection strategy. A complex strategy might even choose to sweep only some of the chunks in a particular generation.

The sweep algorithm starts at the top of a chunk. An object always starts at the first address in the chunk. When chunks are created, they are initialized to contain at least one object. As they are filled with live objects and go through the reclamation process, they are always in a consistent state. Each object in the chunk is either a live object, a dead but not reclaimed object, or an empty slot on the free list. If an object is on the free list, a single bit is set to signal this state. Objects that do not have this bit set are either alive or ready to be collected. The former are left untouched but the later are moved into the free list. If an object is not on the free list, we can determine its size by using the field, which every object contains, that points to its class structure. The class structure has a field giving the size of an instance of that class in bytes. This size is added to the number in the padding field to determine where the next object in the chunk begins. When an object is garbage, we use the fact that it is never less than 16 bytes to place the free list pointers and size field over its old data fields. All consecutive garbage objects and slots in the free list are

combined to make the largest possible free list slot.

When an object is reclaimed during the sweep phase, its class is checked to determine if it needs special treatment. Instances of the classes `Array` and `String` contain pointers to memory that was allocated directly by a call to `malloc()`. These fields must be returned by calling `free()`. If other special base classes are added to the Dee system, they may also have to be reclaimed in this manner.

Promoting Objects

Right after a live object is marked, its age field is also incremented. If the age reaches a preset threshhold, the object is promoted. The first step in the promotion process is to allocate space for the object in the next older generation. A function called `NewGenSlot()` is called to find this new space. `NewGenSlot()` is very similar to `Oalloc()`. They differ in that `Oalloc()` only allocates space from the yongest generation while `NewGenSlot()` only allocates from generations older than the first. `NewGenSlot()` automatically determines the generation of the object passed to it by examining its `Gen` field. It uses this information to decide which generation to allocate from. The correct amount of space needed in the new generation is determined by using the `Class` field in the object being promoted. The class structure has a field that contains the number of bytes any instance of that class will require.

Before copying the object into the new generation, a temporary variable is used to save the amount of padding contained in the slot returned by `GenNewSLot()`. The amount of padding in an object is completely determined by where the object is

allocated. It would be incorrect to copy the padding from the object at its original location to the new location.

After Space has been allocated in the new generation, the object is then copied into this new space. The amount of padding, saved in a temporary variable, is copied into the new slots `Pad` field. At this point, two copies of the object exist. We must be sure that the old copy will be reclaimed during the sweep phase, and that the new object will be left alone. This is acomplished by marking the new location as live, and leaving the old copy unmarked. In addition, we must adjust the `Gen` field in the new copy to reflect its home in a new generation.

The final phase of promoting an object is to readjust all live objects in the system that point to the old copy. This is acomplished by setting a bit in the old object that signals that it has been promoted. This bit, named the `Promo` bit, is used in a second traversal of live objects. After setting the `Promo` bit, a pointer to the new location of the object is placed into one of the fields of the old copy. This is harmless because we have a new copy of the object. In the second traversal, every pointer is dereferenced to determine if it points to an object with its `Promo` bit set. If the `Promo` bit is set, the pointer is overwritten with a copy of the new address of the object found in the old copy's `InstVars[0]` field. This technique of pointer updating is a modified version of an algorithm found in [**?**].

The second traveral of all live objects is only carried out if at least one object has been promoted.

Performance and Tuning

Because the Dee system is in its infancy, it lacks a large application base on which the garbage collection subsystem could be tuned to optimum performance. This section contains an explanation of all the parameters that might help a future programmer optimize the garbage collector.

The size of a chunk can be altered by changing the constant `CHUNK_SIZE`. It is currently set to the size of a page on the SPARC stations. If large objects are being created it makes sense to increase the size of a chunk to several times that of a single page. This variable might also change if the Dee system is ported to new hardware with different characteristics.

Free slots placed on the free list are always added to the beginning of the list. The list is always searched from begining to end, employing a *first fit* strategy, when allocating new slots. `AddToList()` could be changed to maintain the free list in a some other order. The allocation routine `Oalloc()` could be changed to dole out slots in a different fashion, such as *best fit*.

The most significant factor affecting the garbage collector performance is determining when it is invoked and on which generations. The more often the collector is run, the less memory is used by the program. This is because garbage is recycled sooner. Frequent collections have the disadvantage of slowing down the program.

Having a generational collector allows us to call the collector frequently without causing noticible pauses in program execution. Each time the collector is called we

can choose to sweep only a portion of total memory. Currently, the Dee run-time system only calls the garbage collector when the first generation is completely full. Before a new chunk is added to the first generation, the collector is always run. This causes a cycle in the growth of the first generation. When the generation is full, it is swept and then its size is increased by one chunk. Because the first generation contains young objects, a very large percentage of the memory it contains is reclaimed. It then fills up again and repeats the cycle. This has the disadvantage of allowing a program that runs for a long time to indefinitely increase the amount of memory it consumes.

A different approach would be to call the collector on the first generation whenever memory usage reaches a predetermined threshold. Each time the routine to allocate new objects (`Oalloc()`) is called, it would keep statistics about memory usage. The threshold could be set to eighty percent capacity of the total amount of memory available in the generation. If a suitable amount of memory is not reclaimed, another chunk could be added. This algorithm would have the advantage of keeping the amount of memory allocated to the program to a minimum but would also result in an increased number of calls to the collector. The increased number of calls to the collector would slow down the program.

Everytime the collector is called, it must make a decision about how much memory is to be swept. In the current Dee collector, the first generation is swept every time, the second is swept every seventh time and the third is swept every seventeenth time. It is easy to adjust the collector to sweep different generations at different

intervals. Some simple tests revealed that increasing the frequency of sweeping caused noticeable pauses in the program with only minor decreases in the total amount of memory consumed. Using the above intervals prevents the collector from sweeping the second and third generation in the same invocation to often. The choice of the sweeping intervals was based on estimates and a small number of tests. After larger, more sophisticated Dee programs are written, better test may cause them to be adjusted.

Ungar and Jackson [**?**] have shown that objects which live long enough to reach the oldest generation become garbage at a very slow rate. For this reason, the oldest generation in the Dee collector is never swept. Turning collection on for the oldest generation in a few test programs showed no reduction in the number of chunks allocated to the final generation over runs with collection turned off. This policy can result in memory leakage. It would be better to sweep the last generation, but only at very distant intervals.

The current version of the Dee garbage collector maintains a large number of statistics. These are used for debugging and for tuning the algorithm. At the end of every Dee program a few of these statistics are printed. Figure **??** shows the last portion of the output of the Primes program. This program calculates the Prime numbers up to 2000 using Eratosthenes' sieve. It creates a *filter* object for each prime number.

The number for `Bytes Oalloced` is the amount of memory that would have been allocated to the program if it had had no garbage collection. If a compiler option

```
1951
1973
1979
1987
1993
1997
1999

Bytes Oalloced = 1661204, Bytes Malloced = 80864
Chunks per Gen: 58, 5, 5, 8
```

Figure 2.18: Code generator statistics.

is added to Dee to turn off garbage collection, it can be implemented by placing a simple test at the beginning of the `Oalloc()` routine. If "No collection" is flagged, then `Oalloc()` would simply call `malloc()` to get the number of requested bytes.

The amount for `Bytes Malloced` is the actual number of bytes allocated by calls to `malloc()`. It is the total number of bytes dynamically allocated at run-time. This number is also the number of chunks allocated multiplied by the size of a chunk.

The `Chunks per Gen` data are the number of chunks allocated for each generation.

2.8 Base Classes

Base classes are classes that contain a component which is not a pointer to another class. This component comes in the form of a piece of data that can be manipulated directly by the processor. Base classes also differ from regular classes in that they have a very intimate relationship with the code generator and the run-time system.

Even though Dee is a pure object oriented programming language, it must contain

```
struct Object {
ClassPtr Class;
struct GCFields Status;
union {
   int Int;
   ObjectPtr Bool;
   char *String;
   double Float;
   unsigned char Byte;
   ArrayPtr ArrayBody[1];
   ObjectPtr InstVars[1];
   int GCStuff[2];
   } Tag;
};
```

Figure 2.19: The run-time data structure for an object.

data in a form that is manipulatable by the CPU. Regular objects contain instance variables that are pointers to instances of a particular class. Base classes have only one instance variable, and this variable is of a special nature.

Figure **??** shows the C data structure that represents an object at run-time. The first five fields in the variant part of the structure are used to implement the data that is unique to base classes. For example, if the structure is an integer at run-time, the `Int` field will hold its integer value. This `Int` field can only be manipulated by `special` methods of the class `Int`. The fields for float and byte are used in the same manner. The fields for string and boolean classes are used a bit differently.

The field `String`, used only by the string class, points to an array of characters in memory. This field is created by the run-time function `CreateInstance()` or by a special method of the string class. These functions use `malloc()` to allocate the

```
void Int__plus( osp )
   int osp;
{
   os[osp] = CreateInstance( _ClassTableIndex_Int, hwm );

   os[osp] -> Tag.Int =
      os[osp+1] -> Tag.Int + os[osp+2] -> Tag.Int;
}
```

Figure 2.20: The `special` Dee method to add two integers.

memory that actually holds the characters of the string. It would have been difficult to place the characters of the string directly in the structure of the object because strings in Dee are of variable length. They grow and shrink automatically to fit the size of the string that they hold at the moment.

The field `Bool` can only point to one of two objects: the false object or the true object. The two objects are created before any other Dee statements are executed. In order to test the truth value of an object of class `Boolean`, we test the `Bool` field to see which of these two objects it points to.

Using a variant record to hold the different possible values associated with each type of base class saves memory, but at the cost of adding one restriction to the inheritance rules. It is not legal to inherit from a base class in Dee. If this were allowed, it would be necessary to have distinct slots for all the different base class data values.

Figure **??** shows the special method used to add two integers together. The first C statement creates a new instance of the `Int` class and places it in the result slot

of the current Dee stack frame. The second C statement adds the `Int` fields of the object passed to the method and the object in the self field of the current Dee stack frame. Their sum is assigned to the `Int` field of the integer object just placed in the result slot. This is typical of how the special methods of the bases classes actually manipulate data contained in the base class data fields. Appendix B contains the complete Dee and C source code for the base class `Int`.

One should also note the name of the special method in the above example. As mentioned in Section **??** on the parser, the infix operators, like "`+`", are transformed into regular identifier names prefixed with an underscore. This allows programmers to follow the same method naming convention used when the compiler generates stubs for all the other Dee special methods.

Chapter 3

Conclusion

3.1 The Dee System

In this thesis we have explored the Unix implementation of the Dee System. Because it is rare in the literature, we have taken the time to build both hand-coded and generated scanners in order to get an accurate comparison. We have demonstrated how a carefully planed AST definition can be used to allow a group of programmers to work independently on different parts of the same compiler. A lack of discussion in the literature was enhanced by a careful description of code generation from an AST to C source code. Innovative techniques for exception handling and browsing were presented. And finally, one of the first in-depth discussions of a mark-and-sweep generational garbage collector was presented. Unlike other generational collectors, ours included the ability to dynamically increase the size of each generation.

The Dee System can be judged on two different criteria. The first, and most

pertinent to this thesis, is an evaluation of the compiler and its environment. This would include the factors that determine how usable this particular implementation of the system is. The second criterion would try to determine how successful the Dee language is at solving software problems. An excellent implementation of an inadequate idea is of little value to the field of Computer Science. I conclude this thesis with a discussion of the different components of the compiler and suggest ways in which they might be improved.

Creating a hand coded scanner and parser would slightly enhance the performance of the compiler, but the time it would take to do this could be better spent elsewhere. If any change to these components is undertaken, it would make the most sense to improve their error reporting capabilities.

Code generation is the portion of the compiler with the most room for improvement. Because generated code is so difficult to debug, the current version makes no attempt to optimize at all. Future versions could produce better C code by implementing some very simple improvements such as peep hole optimization and removal of unreachable code.

Generating code directly into object format and avoiding the C intermediate step would improve performance significantly. This is a very difficult step and would most likely tie the Dee System to a particular piece of hardware. A better alternative might be to use a machine-independent back end. The GNU team has a back end which produces code for a large number of different CPUs. Documentation for the software is nonexistent, but the source code is freely available. If documentation does

become available, this software might serve as an inexpensive but highly portable and efficient way to compile directly from Dee code into object code.

The garbage collector could also be improved in several ways. Currently, it is quite efficient in time and memory reclamation, but expends the very large overhead of one word per object. It would not be difficult to get this down to a few bits per object. As Goldberg points out in [**?**], modern garbage collectors should be able to eliminate, almost entirely, any per object tag fields.

I believe the implementation described in this thesis is a surprising success. There is no end of improvements that can be made, but given the fact that it was completed by only a few master's students working under one supervisor, we have achieved impressive results. The Dee system works. It is the first implementation of a strongly typed programming language that supports the object oriented paradigm, exception handling, multiple inheritance, automatic garbage collection and a sophisticated browser, that we are aware of. We have successfully created a small class library and run several small applications.

In order to judge the success of the Dee language, many more applications will have to be created. As the compiler undergoes more and more use we will be better able to understand its strong points and weaknesses.

3.2 Related Work

The work that is closest to the Dee project is the *Self* project being carried out at Stanford University. David Ungar and Craig Chambers have created a Smalltalk-like language and environment, which supports the object oriented paradigm. Unlike Dee, Self is dynamically typed and compiles directly to machine code. For more information about Self see [**?**].

Much work is also being done in the field of garbage collection. Work on improving the efficiency of allocating records is being done by Appel [**?**]. One issue that was not discussed in depth in this thesis is locality of memory access during garbage collection. Work is being done in this area by Wilson, Lam and Moher [**?**].

Bibliography

[Ams91] Jonathan Amsterdam. Taking exception to C. *Byte*, August 1991.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, pages 171–183, February 1989.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BL70] P. Branquart and J. Lewis. A scheme of storage allocation and garbage collection for algol-68. In *Algol-68 Implementation*. North-Holland Publishing Company, 1970.

[Boo83] Grady Booch. *Software Engineering with Ada*. Benjamin Cummings Publishing Co., 1983.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18:807–820, September 1988.

[Che92] Benjamin Cheung. Dee: An object oriented programming environment and its implementation. Master's thesis, Concordia University, 1992.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Trans. Programming Languages and Systems*, pages 532–553, October 1983.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Comm. ACM*, pages 655–657, 1960.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *OOPSLA'91*, pages 1–15, 1991.

[Don90] Christophe Dony. Exception handling and object oriented programming: towards a synthesis. In *ECOOP/OOPSLA '90 Proceedings*, pages 322–330, October 1990.

[FH91] Chrisopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26:29–43, October 1991.

[FL88] Charles N. Fisher and Richard J. LeBlanc. *Crafting A Compiler*. The Benjamin Cummings Publishing Company, 1988.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *ACM Conf. on Programming Language Design and Implementation*, pages 165–175, June 1991.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The language and its Implementation*. Addison-Wesley, 1983.

[Gro91] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.

[J.60] McCarthy J. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, pages 184–195, 1960.

[Les75] M. E. Lesk. Lex—a lexical analyzer generator. Computer Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[LH83] Henry Lieberman and Carl Hewitt. A real time garbage collector based on the lifetimes of objects. *Comm. ACM*, 26:419–429, June 1983.

[LS81] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Trans. Soft. Engrg.*, SE-5:546–558, 1981.

[Mar70] S. Marshall. An algol-68 garbage collector. In *Algol-68 Implementation*. North-Holland Publishing Company, 1970.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988.

[ND81] K. Nygaard and O-J. Dahl. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, chapter IX, pages 439–493. Academic Press, 1981.

[Sak88] M. Sakkinen. On the darker side of c++. In *European Conf. on Object Oriented Programming*, pages 162–176. Springer, 1988.

[Som89] I. Sommerville. *Software Engineering.* Addison-Wesley, third edition, 1989.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88 Proceedings*, pages 1–17, September 1988.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, September 1984.

[Wai86] W. M. Waite. The cost of lexical analysis. *Software: Practice and Experience*, pages 473–488, May 1986.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective Static-graph reorganization to improve locality in garbage collected systems. In *ACM Conf. on Programming Language Design and Implementation*, pages 177–191, June 1991.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. *Proc. ACM Conf. on LISP and Functional Programming*, pages 87–98, June 1990.

Appendix A

The Abstract Syntax Tree

```
/*
Dee AST struct definitions.
*/

#ifndef _DEEDEFS_
#define _DEEDEFS_

#define NIL NULL

typedef enum { FALSE, TRUE } Boolean;
typedef int HashIndex;
typedef char *StringPtr;

/* Loop types */

typedef enum { Infin_l, While_l, Until_l } LoopType;

/* Literal Number types */

typedef enum { Int, Float, Byte } NumberType;

/* Mode for local variable */

typedef enum { Param, Result, LocalVar } LocalModeType;

typedef enum { IGNOREATTR, FROMSELF, FROMPARENT,
```

```
                FROMCLAUSE } AttriSrc;

typedef enum { MethodM, ConsM } MethodType;

/* A method body is a ``from'', abstract, concrete or an Instr */

typedef enum { BodyUnknown, BodyFrom, BodyAbs, BodyConcrete,
                BodySpecial } BodyType;

/* An ident node can be a local, inst var, handler local or method */

typedef enum { IdenLocal, IdenInstVar, IdenMethod,
                IdenCons, IdenHandlerLocal } IdenType;

/* Abstract syntax tree definitions. */

typedef enum {
    List, Class, Type, Var, Method,
    Local, Assign, If, IfPair, DoLoop, Loop,
    Apply, Iden, Break, Continue, Nil,
    Attempt, Handler, Signal, Bool, Number,
    String, Null, Undef, Signature, SymTab, CTemp
    } ASTNodeType;


typedef struct ASTNode *AST;

struct ASTNode {
   ASTNodeType NType;
   int Column, Line;

   union {

      /* Lists of nodes are reprsented with List nodes. The
         empty list is represented by NIL. */

      struct {
         AST Node;
         AST Next;
      } List;

      /* A Class node describes an entire class. */
```

```
struct {
   HashIndex ClassName;      /* Name of the class */
   StringPtr ClassComment;   /* Comment following class header */
   AST ClassParamList;       /* List of Signature nodes */
   /* List of the actual classes corrspnding to formal class params */
   AST InheritList;          /* List of Class for inherited classes */
   AST ExtendList;           /* List of Class for extended classes */
   AST InvarList;
   AST AttributeList;
   AST Ancestors;            /* list of all ancestors of this class */
   AST Uses;                 /* The classes of all variables used in */
                             /* stmts of all methods */
   Boolean ClassHasSpecial;  /* true if the class has any special methods */
} Class;

/* A type has a name and a list of arguments, which are themselves
  types. E.g. Array[Table[Int String]]. */

struct {
   HashIndex TypeName;     /* Type name */
   AST TypeArgList;        /* List of Type containing arguments */
} Type;

struct {
    HashIndex SigId;
    AST SigType;
    AST SigOriginalType;   /* Never altered by type substitution */
    int StackOffset;
} Signature;

/* Instance variable descriptor. */

struct {
   StringPtr VarComment;   /* Comment following variable */
   AST VarType;            /* Type node giving the type of the
                              variable */
   Boolean VarPublic;      /* True if this is a public variable */
   AttriSrc AttributeSource;
   AST SourceClass;
} Var;

/* Method descriptor. */
```

```
struct {
   Boolean MethPublic;     /* True if this is a public method */
   MethodType MethKind;    /* One of method or cons */
   HashIndex MethName;     /* Method name */
   StringPtr MethComment;  /* Comment following header */
   AST Result;
   AST MethOriginalResult; /* never altered by SA */
   AST MethLocalList;  /* List of Local local var descriptors         */
   AST MethParamList;  /* This is a pointer into the MethLocalList
                          where the parameters start (not sep. list)  */
   AST Require;        /* Require part of a method  */
   AST Ensure;         /* Ensure part of a method   */
   AST Body;           /* List of statement nodes   */
   BodyType MethBodyType;  /* What kind of body does this method have */
   AST DefinedBy;      /* Set by the SA */
   AST ImplementedBy;  /* Set by the SA */
   int LocalCount;     /* Number of local variables */
   int ParamCount;     /* Number of parameters      */
   AttriSrc AttributeSource;
   BodyType FromBodyType; /* The true body type of a from body */
} Method;

/* Local variable descriptor. Local variables include parameters,
  result, self, and declared local variables. */

struct {
   HashIndex LocName;      /* Local variable name */
   AST LocType;            /* Type node giving type of variable */
} Local;

struct {
   HashIndex Id;
   IdenType IdenKind;
   int LocDisp;            /* Stack displacement if a local        */
   AST IdenType;           /* type of this id filled in by the AST */
 } Iden;

/* The next group of nodes represent statements. */

/* Assignment statement: LHS := RHS. LHS is always a local
   variable or self instance var */

struct {
```

```
   AST AssignVar;          /* LHS Identifier node    */
   AST AssignExpr;         /* RHS expression subtree */
} Assign;

/* If statement */
struct {
   AST IfPairList;         /* List of IfPair nodes */
   AST IfElse;             /* List of statements in the else part */
} If;

/* A pair consisting of an expression E and a list of statements S,
   corresponding to "if E then S" or "elsif E then S". */
struct {
   AST PairExpr;           /* Bool expression    */
   AST PairStmts;          /* List of statements */
} IfPair;

/* A controlled loop:
   "from S until E while E do S od". */
struct {
   AST FromStmts;           /* List of initialization statements */
   AST UntilCond;          /* Bool expression */
   AST WhileCond;          /* Boolean expression */
   AST LoopStmts;       /* List of loop statements */
} Loop;

/* Attempt statement:
    attempt S handlers end */
struct {
   AST AttStmtList;        /* List of statements to be attempted */
   AST AttHandlerList;     /* List of Handler exception handlers */
} Attempt;

/* An exception handler:  var:type statements. */
struct {
   AST HandlerVar;         /* Local node for handler variable */
   AST HandlerStmtList;    /* List of statements for handler  */
} Handler;

/* Signal statement */
struct {
  AST SignalExpr;         /* Expression node for exception object */
} Signal;
```

```
      /* An application node can occur either as a statement or an expr. */
      struct {
         AST Receiver;           /* Either an Apply node or an Iden node  */
         HashIndex AttrName;     /* Name of the method in the application */
         IdenType AttrKind;      /* can only be InstVar, Method or Cons   */
         AST AttrType;           /* static class of the attribute         */
         AST ApplyList;          /* List of expressions: the arguments    */
      } Apply;

      /* The following nodes represent expressions. */

      /* The expression "undefined Expr". */
      struct {
         AST UndefExpr;         /* Expression node */
      } Undef;

      /* A boolean literal: either TRUE or FALSE. */
      struct {
         Boolean BoolVal;
      } Bool;

      /* A numeric literal which may be an Int or a Float. */
      struct {
         NumberType NumKind;     /* an int or a float              */
         int IntVal;             /* if int, here's the real value  */
         double DoubleVal;       /* if float,          "           */
         unsigned char ByteVal;
         StringPtr NumVal;       /* String representation of value */
      } Num;

      /* A string literal */
      struct {
         StringPtr StrVal;
      } String;

   } Tag;

};  /* ASTNode */


#endif
```

Appendix B

Class Int Implementation

The Dee source the the class `Int`.

```
class Int

-- The basic class whose instances are integers.

inherits Ring Order Index

public method get
-- temp way to read an int from the keyboard (does not create the int first)
  special

public method print
-- temp way to print an int
  special

public method show : String
-- temp way to print an int
  special

public method maxint: Int
-- Return the largest integer that can be represented.
  begin
    result := 2147483647
  end

public method = (other: Int): Bool
-- Return true iff the receiver is integer equal to the argument.
  special

public method zero: Int
```

```
-- Return the integer value 0.
  begin
    result := 0
  end

public method one: Int
-- Return the integer value 1.
  begin
    result := 1
  end

public method + (other: Int): Int
-- Return the integer sum of the receiver and the argument.
  special

public method - (other: Int): Int
-- Return the integer difference of the receiver and the argument.
  special

public method * (other: Int): Int
-- Return the integer product of the receiver and the argument.
  special

public method / (other: Int): Int
-- Return the integer quotient of the receiver and the argument. System
-- exception if the argument is zero.
  special

public method mod (other: Int): Int
-- Return the integer modulus of the receiver and the argument. System
-- exception if the argument is zero.
  special

public method < (other: Int): Bool
-- Return true iff the receiver is integer less than the argument.
  special

public method float: Float
-- Return a floating point number with the same value as the receiver.
  special

public method char: String
-- Return a one-character string consisting of the ASCII character whose
```

```
-- code is the receiver. Exception if the receiver is outside the range
-- [0..255].
  special

public method abs: Int
-- Return the absolute value of the receiver.
  begin
    if self >= 0
      then result := self
      else result := 0 - self
    fi
  end

public method gcd (y: Int): Int
-- Return the greatest common divisor of the receiver and the argument.
--  Exception 101 if either argument is zero.
  var x: Int
      rem: Int
  begin
    if (self = 0) or (y = 0)
      then signal 101
    fi
    x := self.abs
    y := y.abs
    from until y = 0 do
      rem := x mod y
      x := y
      y := rem
    od
    result := x
  end
```

The implentation of the special methods of class `Int`.

```
/*
  Special Dee instructions for class Int
*/

#include "CeeGlob.h"


extern int _ClassTableIndex_Int;
extern int _ClassTableIndex_Float;
```

```
void Int_print(osp)
     int osp;
{
  printf( "%d\n", (os[osp+1] -> Tag.Int) );
}

void Int_get(osp)
     int osp;
{
  scanf( "%d", &(os[osp+1] -> Tag.Int) );
  os[osp] = os[osp+1];
}


void Int__eq( osp )
     int osp;
{
 if ( os[osp+1] -> Tag.Int == os[osp+2] -> Tag.Int )
   os[osp] = true_object;
 else
   os[osp] = false_object;
}


void Int_char( osp )
     int osp;
{
  char s[2] = " ";
  *s = (char) os[osp+1] -> Tag.Int;
  os[osp] = create_string( s, hwm );
}

void Int_show( osp )
     int osp;
{
  char buf[20];

  sprintf( buf, "%ld", os[osp+1]->Tag.Int );

  os[osp] = create_string( buf, hwm );
}
```

```
void Int__plus( osp )
     int osp;
{
  os[osp] = CreateInstance( _ClassTableIndex_Int, hwm );

  os[osp] -> Tag.Int = os[osp+1] -> Tag.Int + os[osp+2] -> Tag.Int;
}


void Int__minus( osp )
     int osp;
{
  os[osp] = CreateInstance( _ClassTableIndex_Int, hwm );

  os[osp] -> Tag.Int = os[osp+1] -> Tag.Int - os[osp+2] -> Tag.Int;
}


void Int__mul( osp )
     int osp;
{
  os[osp] = CreateInstance( _ClassTableIndex_Int, hwm );

  os[osp] -> Tag.Int = os[osp+1] -> Tag.Int * os[osp+2] -> Tag.Int;
}


void Int__div( osp )
     int osp;
{
  os[osp] = CreateInstance( _ClassTableIndex_Int, hwm );

  os[osp] -> Tag.Int = os[osp+1] -> Tag.Int / os[osp+2] -> Tag.Int;
}


void Int__mod( osp )
     int osp;
{
  os[osp] = CreateInstance( _ClassTableIndex_Int, hwm );
```

```
  os[osp] -> Tag.Int = os[osp+1] -> Tag.Int % os[osp+2] -> Tag.Int;
}



void Int__ne( osp )
     int osp;
{
 if ( os[osp+1] -> Tag.Int != os[osp+2] -> Tag.Int )
   os[osp] = true_object;
 else
   os[osp] = false_object;
}


void Int__lt( osp )
     int osp;
{
 if ( os[osp+1] -> Tag.Int < os[osp+2] -> Tag.Int )
   os[osp] = true_object;
 else
   os[osp] = false_object;
}



void Int__gt( osp )
     int osp;
{
 if ( os[osp+1] -> Tag.Int > os[osp+2] -> Tag.Int )
   os[osp] = true_object;
 else
   os[osp] = false_object;
}



void Int__ge( osp )
     int osp;
{
 if ( os[osp+1] -> Tag.Int >= os[osp+2] -> Tag.Int )
   os[osp] = true_object;
 else
```

```
   os[osp] = false_object;
}


void Int__le( osp )
     int osp;
{
 if ( os[osp+1] -> Tag.Int <= os[osp+2] -> Tag.Int )
   os[osp] = true_object;
 else
   os[osp] = false_object;
}




void Int_float( osp )
     int osp;
{
  os[osp] = CreateInstance( _ClassTableIndex_Float, hwm );
  os[osp] -> Tag.Float = (double) os[osp+1] -> Tag.Int;
}
```