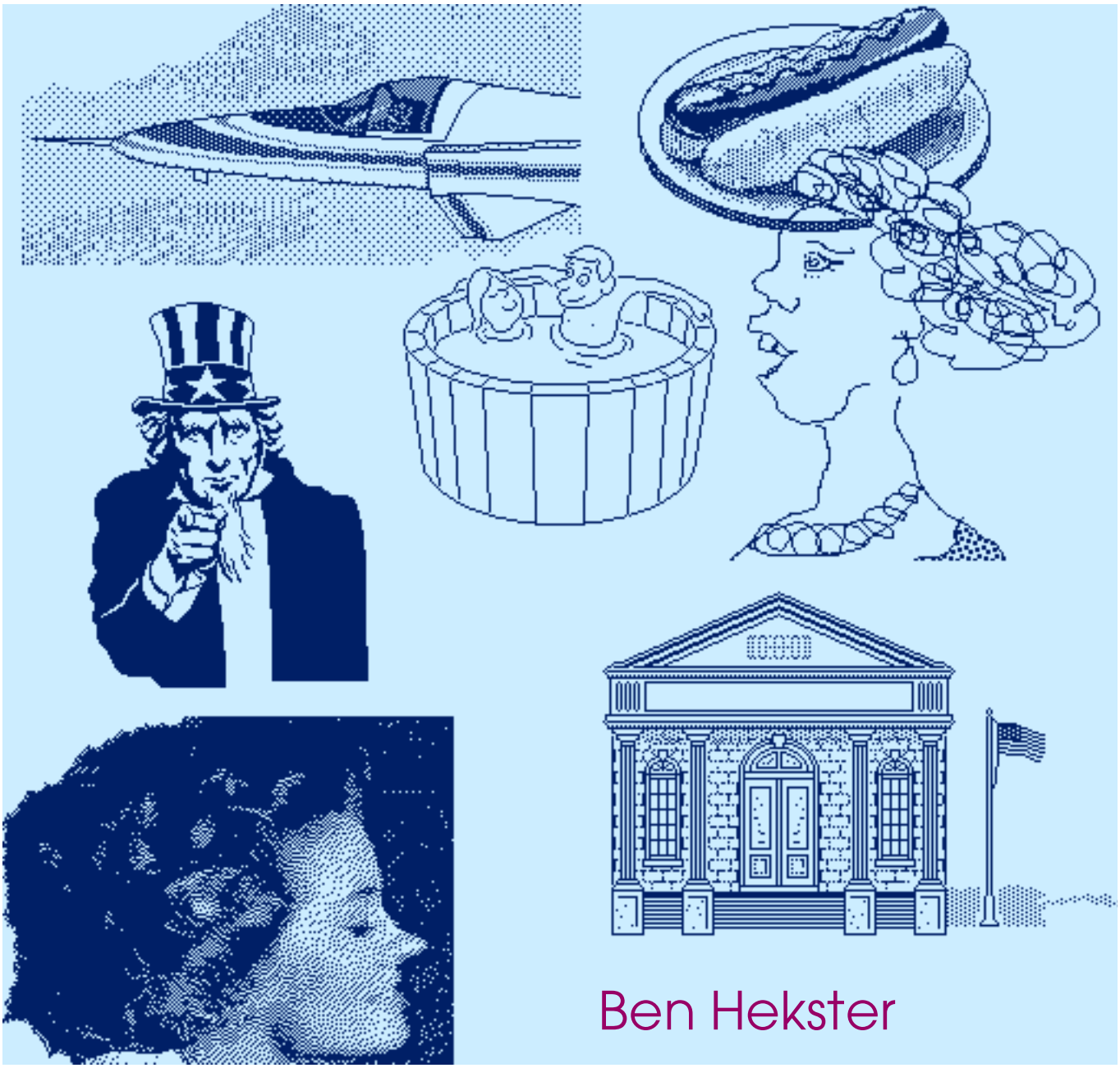
The Programming Language GRUNT-92



Ben Hekster

1

Proem

Something stops me from seeing too clear
A misty haze floats 'round the room
The murmurs make it hard to hear
The voices and the words
—*Your Name (Has Slipped My Mind Again)*, Ultravox [Rage in Eden]

Before you is the report of design theory and implementation of the *Grunt-92* programming language, which is the result of the Master's thesis (*1000-uurs opdracht*) for the Electronic Engineering degree of the University of Twente.

The language

Grunt-92 is actually less a language and more an integrated 'object system' that supports the construction of language operations. The object system is based on a very small number of well-chosen primitive principles: the *object*, the *metaobject*, and the *message*. An object responds to messages, message response is determined by the metaobject, which is itself an object. The metaobject is the embodiment of the notion that data and the representation and behavior of this data are distinct. These trilogy is flexible enough to allow the construction of more traditional higher-level constructs, and hopefully, new untried mechanisms as well.

There is, however, a *canonical grammar* which corresponds very closely to these object system primitives. This makes it a pretty low-level language, although it does actually provide some additional support above the strict primitive level. Its close relation to the primitive mechanisms makes it useful as a language for expressing higher-level constructs. Because the object primitives and the canonical grammar correspond so closely they are often identified with one another— that is, "Grunt-92" refers to both the canonical grammar and the objects system itself. The simplicity of Grunt-92 is, like Smalltalk, due to the *consistent* application of its primitives, and is reflected by an exceedingly simple grammar (with only 21 productions).

Grunt-92 is a reflective language. Language operations are expressed within the system itself. New languages can be introduced by expressing their mechanisms in terms of the system objects.

Unlike other languages, Grunt-92 has measures to improve code efficiency built in from the start. These measures address efficiency at a more fundamental level than so-called 'code optimizers' found in typical compilers. Because they operate within the system they are independent of any particular source or target languages.

The report

The report begins in Chapter 1 with an introduction into the design rationale and philosophy of the language, followed by a description of the canonical grammar in Chapter 2. The grammar is treated so early on because it is used extensively in the following chapters for the representation of higher-level mechanisms. The Grunt primitive object model is explained in Chapter 3. Chapter 4 elaborates on how this object model can be used as a basis for building more complex mechanisms. Chapter 5 gives details of an 'external representation', which is an external 'image' of objects in the system. Chapter 6 explains the workings of the Grunt interpreter which executes code within the system, whereas Chapter 7 explains how code generators work which generate code to be executed outside of the system. Finally, Chapter 8 concludes with a summary of this work.

typographical conventions

Grunt language code is written in Helvetica type. Other source code, for instance, that used in the implementation of Grunt itself is written in `Courier` type.

> Text set off in a slightly smaller size from the body are annotations, typically highlighting or elaborating on design and implementation alternatives or differences with other languages.

▲ Suggestions for future extensions

History

As its name suggests, Grunt-92 is an evolution of earlier versions of the language. The original Grunt language (also referred to as *Classic Grunt*) was developed in 1990 as the assignment for the course in Compiler Design (*Vertalerbouw*) [Hekster90] and used the TCGS compiler-generation system required for that course. Similarly, Grunt-91 was the assignment for the Special Topics in Compiler Design (*Bijzondere Onderwerpen Vertalerbouw*) course [Hekster91], but was written in C and used more widely available compiler-generation tools based on `lex` and `yacc`. The compiler ran on both Sun workstations and Macintosh computers and generated code for the Motorola 68000 CISC family and tentative code for the Motorola 88100 RISC (the latter could never be verified because there was no hardware available to run it on).

Although the design of the Grunt languages has benefited ideas from many existing languages, it is particularly strongly influenced by and owes its name to the visionary Smalltalk-80 languages. I believe that Smalltalk remains in a class of its own because of its conceptual simplicity and consistency. The name "Grunt" is a tongue-in-cheek reference to the rudimentary nature of primeval personal communication and a hint to the state of the original language relative to Smalltalk.

Grunt-92 has undeniably and unabashedly been supported very much by knowledge developed by others. In particular the language uses, and where appropriate tries to improve on, concepts derived from C in "the new testament" [Kernighan88], C++ [Ellis90], the "pink book" Smalltalk-80 [Goldberg89] (which has all the information of [Goldberg83] except implementation details) and Object Pascal [Tesler85]. In fact, these works are used so often that the explicit author/year-of-publication reference is usually omitted when it is clear which language is being discussed— this generic reference is presumed to suffice.

Acknowledgements

I would like to take this opportunity to thank the members of my 'afstudeercommissie', for giving me the opportunity to explore, make my own mistakes, and hopefully correct them— and, in particular, for their help beyond the call of duty: dr. A. Nijmeijer and dr. H. Alblas, and prof. H. Brinksma.

August 1992

Ben Hekster
Tankelanden 5
7542 DR Enschede
053-764091
heksterb@cs.utwente.nl
heksterb@apple.com

C

Contents

Printed in 10-point New Century Schoolbook

Created on the Apple Macintosh with Microsoft Word and Claris MacDraw Pro
Printed on Monday, October 2, 2000

2

Introduction

Benedick. Your niece regards me with an eye of favour.
Leonato. That eye my daughter lent her. 'Tis most true.
Bene. And I do with an eye of love requite her.
Leon. The sight whereof, I think, you had from me, from Claudio, and the Prince.
Bene. Your answer, sir, is enigmatical.
—*Much Ado About Nothing*

This chapter gives an exposition of the reasoning behind Grunt-92's design, and a hint of object-oriented concepts and terms— just enough in fact to bring the reader up to speed with the more detailed information in the following chapters. It explains that "Grunt" is more of a *system* of objects that happens to implement a compiler, rather than a compiler implemented in an object-oriented way.

2.1. Rationale

Why another language? There were a few reasons behind the development of Grunt-92:

- The SETI *vakgroep* of the University of Twente's Computer Science department wanted to gain experience with compilers for object-oriented languages
- SETI wanted a sort of 'object language environment' support in which students could create their own languages for a laboratory assignment in the Advanced Topics in Compiler Design course

So there were no particularly specific requirements on the function or the purpose of the language. In the absence of such, I added some guiding principles of my own:

- I wanted to learn about different object mechanisms and implement them in a language of my own
- to create a language that unified as many as possible of the novel and advanced object-oriented techniques, while
- maintaining efficiency in object code

I do believe this last point to be very important. It is one thing to design a 'prototype' or 'research' language to develop and evaluate new concepts. It is quite another to do so in a way that it is actually worthwhile and useful in practice. My justification for introducing this criterion right from the start as one basic to the design of a language, is the ubiquity of commercial or 'production' software written in 'efficient' languages (e.g., C, C++) compared to that written in what are generally held to be theoretically purer languages (e.g., Smalltalk or Eiffel).

2.2. Philosophy

The basic tenet guiding the design of Grunt-92 is the application of object-oriented design on the language and the compiler itself.

Consistency in a programming language means the adoption of usually a small number of axiomatic orthogonal principles as its basis. When these principles are combined they need to span a sufficiently large part of the language mechanism domain so that additional *ad hoc* constructs are not required to support less basic functions.

Reification is defined by Webster's *Ninth New Collegiate* as "the process or result of regarding something abstract as a material or concrete thing". In the sense of objects, this means the explicit expression of abstract or formerly unidentified concepts in terms of objects. Reification requires a much closer look at the compiler's inner mechanisms and expressing them in terms of objects. The difficulty in this process lies not just in identifying these objects, but also in determining which object is in fact responsible for which activity. Reification forces all the information that is gleaned but usually stored away by the compiler to the surface, and makes it available to users of the language, who could be expected to make good use of it. A specific example is type information, including both name and structure information.

By applying object-oriented design techniques to the language, the features of the language are themselves expressed as classes in a hierarchy. An elementary application of this principle was demonstrated by [Wegner87] in his expression of the object language taxonomy in terms of a class hierarchy. Not only does such an explicit description of programming languages enable users of programming languages to better understand the nature and interrelationships of the mechanisms they are using, but, provided the mechinisms are made available to the user, to be more selective in choosing the ones that they want.

Once the system is reified, the consistent application of principles makes the system open and extensible and allows one to apply the same techniques of prototyping and incremental refinement to the compiler and the language itself. This is good, because open systems have often shown themselves to be more responsive to new developments and are much more successful in standing the test of time. Users do not have to await new releases of the language. A danger of open systems, of course, is the emergence of diverging standards.

Grunt-92 is like Smalltalk in that it reifies the compiler (or interpreter) within the system. Because anything in the system is available for modification or extension by the user, this has two important consequences: first, it is possible to add and modify mechanisms to the language later on; and second, all the information available to the compiler is also available to the user. In fact, this last point proves actually to obviate the need for defining many new mechanisms.

2.3. Compilation

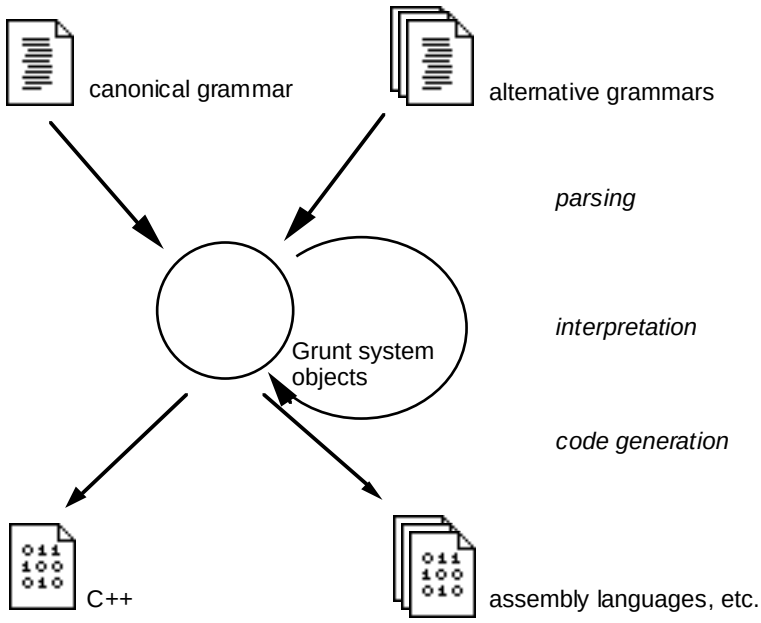
Grunt-92 is a system of objects that can be used to compile programs. Although compilation is a function of the object system, it is not necessarily a fundamental one. What the system does depends only on what messages are sent to which objects.

Compiling a user program involves using objects in the system to create new objects which represent the program being compiled. Because the entire process of compilation happens within the system, it is possible to isolate and reify independent 'stages' of the compilation process that are traditionally combined into one indivisible operation. This means that the results from each individual stage become available to all the others, so that the progress of information during the compilation process need not a 'one-way' flow. This in turn has many advantages, as will hopefully become clear throughout this report.

In Grunt-92, compilation consists of three identifiably separate steps: *parsing*, *interpretation*, and *code generation* (Figure 1.1). Their roles will be highlighted in this section. Because every object in the system could conceivably play a role in compilation, there is, however, no single identifiable 'compiler' object. 'Compilation' means, simply, using the system to generate code. Whenever a reference is made to a 'compiler', all of the system involved in compilation is meant.

Although the process of compilation is represented as a staged process, not all parts of a compiled program necessarily pass through all three stages, nor are

Figure 1.1
phases of compilation



the stages mutually chronological. Some parts are not code-generated because they are resolved by the interpreter, and indeed, some parts are not parsed because they are part of the language being compiled and hence are already present in the system. Furthermore, it is completely possible for one part of a program to have completed code generation, while another part has yet to be parsed.

parsing

A **parser** (also commonly known as a *front end*) object translates a (usually human-readable) *source* language into object forms that can be understood by the Grunt-92 object system. New languages may be made known to the system in the form of parser objects, thereby providing new ways to write programs in Grunt. In other words, the system can be programmed in potentially many different languages. There is a **canonical parser** provided with the system which can be used to write programs, including, for example, other such parsers.

A parser translates the statements of the language into new Grunt objects and messages that can be executed by the object system. This situation is conceptually very similar to that of the UNIX kernel and its 'surrounding' programming shells. The shells are what a user normally sees and uses to write and run programs, but the actual work is done by the operating system kernel which the shell invokes appropriately.

Because a Grunt-92 parser is itself an object, its methods can be overridden and reimplemented, it can even be invoked from within a program, and anything else that could possibly be done to an object can be done to it. Although it would in general be possible to modify a language simply by overriding a parser's methods, the canonical parser was written in C with `lex` and `yacc`, and implemented as a single Grunt method, making it somewhat difficult to selectively modify that grammar.

The main reasons why the canonical parser was implemented in this object-unfriendly way is that it was easier to use already available parser-generator tools to bootstrap the rest of the Grunt-92 object system. It would have been more difficult to implement the parser in the very language it is meant to implement before the language even exists. However, doing it this way does demonstrate the ability to use code written in other languages (*external code*),

and the manner whereby students can use these widely available and more generally applicable compiler-generation tools to implement their own compilers.

interpretation

The second and most complex phase of compilation is, for lack of a better word, **interpretation** of the created messages. Interpretation involves the evaluation of as many messages and the concomitant elimination of as much redundant information from the user program as possible. Because Grunt-92 expresses everything down to a very low level in terms of objects and messages, it becomes especially important that as many of these objects as possible are 'resolved' before they become part of the final program. Grunt's interpretation of messages is like Smalltalk's message interpretation, except that since Grunt codifies the notion of 'ignorance' (called *volatility* in Grunt parlance) of the state of an object, certain messages cannot be interpreted. Non-interpretable messages are candidates for code generation in the final phase.

A commonly occurring application by compilers of this principle is constant expression evaluation. However, Grunt interpretation goes much further than this. Variable names, for instance, are also suffused into the object nature of parsed code, and unless special care there is nothing to prevent them from being generated into the final program. Sometimes this is actually desired. Moreover, very fundamental operations such as message lookup are expressed in terms of messages and objects, and it is obviously essential that the interpreter take care of as many of these as possible.

So, where this is usually seen as an 'optimizing' step in compilation, in Grunt-92 it is really an essential quality of the language which permits the practical formalization of compilation within the language itself.

code generation

Finally, a **code generator** (also known as a *back end*) is the step that converts possibly already interpreted messages and objects into a form that can be executed by the target system. Because the interpreter is in theory itself capable of executing the entire program, code generation can be regarded as an optional step, that is capable of greatly speeding-up the execution of the program. As it turns out, code generation can actually be viewed as a 'special case' of interpretation, the difference being that while an interpreter evaluates messages immediately, a code generator generates code immediately that evaluates the messages later.

Two extremes in compilation can be identified. In one extreme it is theoretically possible that the entire program can be resolved by the interpreter at 'compile-time', so that no output code is generated at all. This is also seen to happen with highly optimizing compilers for trivial functions. In the other case, parts of the interpreter itself may be generated into the run-time environment—in effect, the compiler is generating an interpreter for that program. In particular, this would be likely to happen when the interpreter was invoked from within the user's program. Taking this to its extreme, there would be an infinite progression of compilers, generating compilers. All this is possible because the parser, interpreter, and code generator, are just objects in the system and therefore behave no differently from the objects in a user's program.

Obviously for many users, a compiled program *is* the desired result of the compilation, so code generation is in fact an essential step. A user can force certain parts of a program to resist interpretation and pass through to the code generation step.

Again, code generators are (groups of) objects and such can be modified and replaced. New code generators for different target systems can be added to the system. A code generator for C++ code is being written. I chose C++ as a first implementation of a code generator because Grunt-92 (at least the primitive part) itself is written in C++, so that the generated code can be used to 'compile the compiler'. C++ is obviously far easier to read than assembly or machine code (facilitating debugging) and much more portable (the compiler can be used and tested on multiple platforms). This was important because Grunt-92 was developed on a different system than the one for which it was intended to be used. A code generator for an assembly language should in theory allow better code to be generated because Grunt makes all the semantic information of a

program available to the code generator, enabling the code generator to make better use of a target system's particular features.

3

Canonical Grammar

To put it into words
To write it down
That is walking on hallowed ground
—*Sacred*, Depeche Mode [Music for the Masses]

This chapter defines the canonical grammar of the Grunt system. As stated in the introduction, the canonical grammar is simply a straightforward (almost one-to-one) mapping of Grunt primitives, and other grammars may exist or be introduced that implement more complex mappings. Because it is so close to the primitive Grunt level, the canonical grammar is an effective vehicle for expressing these complex mappings. In that light, it is important to realize that most of the information presented here really is particular to the canonical grammar only and not inherent to the Grunt system itself.

The description of Grunt primitives and the complex mechanisms that can be expressed in terms of these primitives follows in subsequent chapters. The canonical grammar is given before the mechanisms that it implements because the code examples used to explain them are written in the canonical grammar. Therefore, this chapter makes no attempt to explain how the mechanisms that it describes actually work in detail. It is probably best initially only to peruse this chapter and try to get a feel for the grammar, and to use it as a reference when necessary.

Because the grammar is so closely coupled to Grunt itself, its straightforwardness and orthogonality are reflected in the simplicity of the grammar. Since most traditional language mechanisms are expressed as Grunt objects, they do not require explicit grammatical support. This means that the canonical grammar is as basic or as powerful as the objects that it uses— it is neither inherently typed nor untyped; class-based, object-based, or neither; and so on.

The canonical grammar provides just enough expressiveness to prevent bootstrapping problems. For example, objects are created by sending message objects, which requires creating the message object— this is why the canonical grammar incorporates a syntax for creating and sending message objects. There is also a syntax for sending multiple messages to the same object, or for sending a message to the result of a previous message send.

The grammar definitions given in this chapter have been taken directly from the `bison` grammar specification file, with the following changes: (1) nonterminals are shown in *italic* and terminals in roman type, (2) alternative productions for the same nonterminal are given on a new line without the vertical bar "`|`" separator, (3) ε-(empty) productions are shown simply as ε, and (4) non-keyword terminals (e.g. "*integer*") are displayed as nonterminals. The grammar is strictly LR(1), i.e., has no shift/reduce or reduce/reduce conflicts.

3.1. Lexical Conventions

Whitespace consists of any consecutive combination of the space, tab and newline characters. Other than that it is used to separate tokens they are ignored.

comments

The two-character sequence // (slash slash) introduces a **comment** on the rest of that line (i.e., until the first newline character or end of file). The rest of that line is completely ignored by the parser:

```
anObject // this is another Integer /* or Boolean
aMessage
```

is equivalent to “anObject aMessage”.

The two-character sequence /* (slash asterisk) introduces a nestable comment which terminates with the corresponding */ (asterisk slash). For example,

```
anObject /* this is the Integer /* or Boolean */ that we */ aMessage /* send the
message to */
```

is also parsed as “anObject aMessage”. As in the first example, /* or */ sequences that appear in an unnested // comment are ignored.

literals

A literal is an objects that are created implicitly, without having to explicitly refer to another object to create it. The type of literals that can be created is limited and unambiguous. Besides block objects (described later), there are four other types of literals:

```
object:
      block
      literal
```

Since the number of different Boolean objects is very limited, Booleans named `true` and `false` are automatically created by the Grunt system when it is initialized— there are hence no Boolean literals. Message expressions also create (message) objects, but since they also cause them to be sent are thus not strictly literal expressions.

An **integer literal** is an optional plus sign + or minus sign –, followed by one or more digits 0 through 9, representing either a signed decimal value, or when followed by r (lower-case alphabetic r) the decimal radix of a positionally represented integer specified by zero or more trailing coefficients.

A single quote ' introduces an integer literal specified by a character string which terminates with the first whitespace. The character literal is interpreted as a radix 256 positional representation integer, where the ‘coefficients’ have the value from their ASCII encoding:

```
'a write.                                  // writes “97”
'abcd asStringBase: 16; write              // writes “16r61626364”
```

'abcd is interpreted as $'a \cdot 16^3 + 'b \cdot 16^2 + 'c \cdot 16^1 + 'd \cdot 16^0$. The message asStringBase: takes an integer radix as an argument and returns a string which is an ASCII representation of the integer receiver in that radix.

The two-character sequence /' (slash single quote) introduces an nonnestable character literal that terminates with the first '/ (single quote slash):

```
/'a
cd'/ asStringBase: 16; write               // writes “16r610a6364”
```

This is mainly useful for denoting character literals containing whitespace.

A double quote " introduces a nonnestable **string literal** which terminates with the next whitespace:

```
"hello write                               // writes “hello”
```

The two-character sequence /" (slash double quote) introduces a nestable character string which terminates with the corresponding "/ (double quote slash):

```
/"hello, world!"/ write
```

As in comments, no parsing occurs within strings:

```
/"one two /* four five "/ asWords, count, write.      // writes “5”
```

since the “/*” is not parsed but remains part of the string.

> Nestable strings are a cleaner solution to the problem of how to represent text containing parseable text. A more traditional method of representing such constructs is through the use of escape characters (such as the backslash in C), but they have the unfortunate side effect that the escape character is usually strewn throughout the inner code, making it appear different from outer-level code. C preprocessor macros, for example, have no terminator character and need to be written on a single ‘logical’ line, requiring backslashes at the end of each ‘physical’ line. The resulting ugliness is partly to blame for having made moderately complex macro programming (potentially a very powerful tool) regarded as bad style. The problem of nesting also occurs very frequently in UNIX shell programming where three different types of quoting need to be carefully and selectively protected from certain levels of interpretation by backslashes, often a nontrivial exercise.

A single quote ' introduces a nonnestable **symbol** literal which terminates with the first whitespace:

```
'hello lexeme, write                                  // writes “hello”
```

The two-character sequence /' (slash single quote) introduces a nestable symbol which terminates with the corresponding '/ (single quote slash):

```
/'hello, world!'/ lexeme, write                       // writes “hello, world!”
```

As in comments, no parsing occurs within keyword literals. The difference between a string and a symbol is that if two symbols have the same value, they are the same object; whereas two identical character strings may be different objects. Symbol objects are used extensively in addressing and indexing operations.

variables

Variables are objects that can be referred to after their creation by means of an identfier.

```
object:
      variable

variable:
      identifier
```

An identifier is any other sequence of non-whitespace characters (including alphanumeric characters and symbols) that is not a comment, literal, operator, and is not a separate occurence of any of the following symbols, which are used as ‘punctuation’ by the grammar:

```
; , . ( ) [ ]
```

The grammar has no keywords that are reserved identifiers. Upper and lower character cases are distinct.

> The rules for what constitutes an identifier are purposely lax because they must be able to identify objects that represent many different sorts of language features, each of which may have very different naming requirements— such as functions, arithmetic operations, types, etc.

The object that is associated with a particular identifier is determined by what is currently in *scope*. The system normally starts up with only the Grunt system dictionary (named systemDict by an association it contains to itself) in scope, but the current scope can be modified by creating additional Scopes. Scoping rules are not made nor enforced by the parser.

Attempting to reference a variable through an identifer that does not have an object association in current scope is an error.

It is possible to define variables with non-identifier names, but they cannot be referred to by the normal implicit variable identifier syntax and instead must be obtained by applying explicitly to the defining scope.

3.4. Messages

The message is the fundamental operation in every object-based language.

```
optMessage:
        cascadedMessage
        ε

message:
        selector optArgumentList
```

Identifiers are used to identify message arguments— the `selector` identifies the first message argument, if present, and is used in the method lookup when the message has no arguments.

```
selector:
        identifier
        ( messageList )
```

A selector may be specified by an identifier, or as the result of a parenthesized expression. A syntactically more consistent syntax would have the selector simply be an `object`, with traditional selectors then corresponding to string literals. The additional quotes needed to specify string literals would be a notational inconvenience (e.g. receiver "literalMessage), which is why the grammatical asymmetry is introduced.

If the message send returns an object by the name result, this object is made the result of the `message`.

```
1 + 2; write
```

takes the result returned by sending count to anArray and writeing it to standard output. If there is no result object in the message, the message expression refers to the message object itself, from which results (of which there may be several) can be explicitly extracted:

```
anArray bounds; `lower write; `upper write
```

The `bounds` message to anArray returns two objects lower and upper in the message, which can be referenced with the meta operator (§2.5). Note that results are simply arguments which are modified by the method.

> In languages that do not directly support messages (or their equivalent) returning multiple results, they can sometimes be simulated by explicitly defining a single compound structure type containing the required individual results. However, this is hardly natural or symmetric with argument passing.

arguments

Messages may have **arguments**, which are objects that are communicated from its sender to its receiver.

```
optArgumentList:
        operatedObject argumentList
        ε

argumentList:
        argumentList identifier operatedObject
```

The first argument is added to the message under the name of the message selector (see above). Subsequent arguments are added under additional specified identifiers. It is possible and legal to specify more than one argument with the same name. The parser automatically also an argument named self to the message which refers to its receiver.

> Other object-oriented languages also treat the receiver object as a special argument (e.g., `this` in `cfront`).

3.4. Compound Messages

Message sequences and cascades often allow one to express certain operations more compactly than if they are written out as a list of individual messages. Individual messages may be **sequenced** or **cascaded** to form more complex constructions.

message sequences

A message sequence is a list of messages to the same receiver.

```
messageSequence:
        messageSequcnce ; operatorList optMessage messageCascade
        messageCascade
```

For example:

```
MetaInteger [] 'instanceObjects
        ; add: "minimum
        ; add: "maximum
        .
```

causes two fields minimum and maximum to be added to the instanceObjects dictionary of MetaInteger.

The result of the `messageSequence` is its receiver; in other words, the results of the messages in the sequence are lost. If an unrecoverable error occurs while sending a message in a sequence, the message is discarded, and execution continues with the next message, if any.

message cascades

A message cascade is a list of messages, the result of each being the receiver of the next.

```
messageCascade:
        messageCascade , operatorList message
```

For example:

```
MetaInteger [] 'instanceObjects
        ; add: "minimum , = pointer-to-meta-of-long-int-pullBack
        ; add: "maximum , = pointer-to-meta-of-long-int-pullBack
        .
```

creates two fields minimum and maximum in the instanceObjects dictionary of MetaInteger, to each which is assigned the value of the variable pointer-to-meta-of-long-int-pullBack.

The result of the `messageCascade` is the result of the last message in the list. If an unrecoverable error occurs during sending of any of the messages in the cascade, any messages remaining in the cascade are discarded.

3.8. Program

A program is simply a list of messages that are sent when it is run:

```
program:
        messageList

messageList:
        messageList operatedObject optMessage messageSequence .
        ε
```

Note that the dot character (or 'period', or 'full stop') "." is a separator rather than a terminator, in contrast to ordinary usage in natural language but in accordance with the other punctuation used in the canonical grammar. In other words, a message ends only in a period if it is followed by another message.

The value of the `program` is the value of the last message expression (`operatedObject optMessage messageSequence`) in the `messageList`. If an unrecoverable error occurs in its evaluation, the rest of the message expression is discarded and its result is null.

The messages in the `messageList` are created and sent immediately. They execute within the context that is established by the Grunt system, and as if in response to a message that was sent to the system itself (so that self is grunt).

Since messages at the outer level are executed straight away, the Grunt compiler can be used interactively:

```
$ grunt                                  # sh command line
> (2 + 3) write.                         // prints "5" on standard output
> [(5 – 2) write] | aBlock.              // create aBlock
> aBlock do                              // prints "3" on standard output
> control-d                              // end of input
$
```

Often, however, input to the Grunt compiler comes from a file which contains messages that create classes and add methods:

```
$ grunt hello.grunt
```

3.5. Operators

Operators are built in to the grammar to express certain basic operations on objects. The messages required to perform these operations are created and sent by the parser. The grammar contains three such operators.

```
operatedObject:
      object operatorList

operatorList:
      operatorList operator
      ε
```

Operators are evaluated left to right. The value of an `operatedObject` is the result of the last (rightmost) operator.

Note that since operators are not messages in the normal sense, they cannot be looked up:

```
1 lookup: '@ ; @ write                   # writes "null"
```

reference operator

The @ 'at' sign or **reference operator** returns a reference to the object it is applied to.

```
operator:
      @
```

The reference is an instance of Reference:

```
( meta-of-long-int pullBack: null ) @
```

is a reference to the object returned by the message within parentheses.

meta operator

The ` backquote or **meta operator** returns the metaobject of the object it is applied to.

```
operator:
      `
```

For example,

```
1 `
```

Is the MetaInteger describing the integer literal 1.

tag operator

The vertical bar | or **tag operator** associates an identifier with the object that it is applied to in the current scope. The result of the tag is simply the object it is applied to.

```
operator:
      | identifier
```

For example,

```
( static allocate: sizeOfMetaInteger ) | meta-of-long-int.
```

associates the name meta-of-long-int to the result of the message within parentheses. Note the semicolon is needed to prevent the tag from applying to the message argument sizeOfMetaInteger.

message scope

The Message in response to which a method is executing is also a scope, containing the message arguments and an association called self to the object that is executing the method. The context in which a message operates is the message to which it responds.

> Note that self is not a pseudo-object like in Smalltalk. There are no pseudo-objects in the canonical grammar other than scope. One of a Message's fields is self, which corresponds to the usage by Smalltalk. Method scopes of instance objects (i.e. those with MetaInstance metaobjects) contain the association between super and the receiver with the metaobject class set to the superclass, which also corresponds to the Smalltalk usage. Because these are not parser-specific pseudo-objects it is possible to implement extensions such as a separate client association in delegated Messages, which solves the inheritance anomaly as described by .

The Message contains associations for the message arguments.

The outer level of a Grunt program, although not technically a method, has a scope automatically created for it.

3.7. Block

A **block** is a list of messages which can be executed. A block does not establish a scope; there is no specific block result and there are no block arguments, although objects may be passed and returned through the scope in which the block executes.

```
block:
      [ messageList ]
```

Blocks are made constant volatile (§3.2) by the parser by default, unless the meta operator is used on the block— this is done to correspond more closely to the typical use, where they do not change once defined and are always generated into the target language.

> Inconstant block variables can occasionally be useful. Consider the instances of Smalltalk's SortedCollection, which contain a sortBlock which defines an ordering on a pair of elements in the collection. The sort order of the collection can be changed by attaching a different sortBlock to it.

3.2. Parser Directives

Since the parser is just an object in the system, parser directives (or *pragmas*) are simply messages to the parser object and do not require any special 'preprocessor' syntax.

For example, the message include: "*filename*" instructs the parser to suspend parsing of the current file while the specified file is parsed, e.g.

```
parser include: /"Initialization.grunt"/
```

`parser` is the name defined by the parser referring to itself. Other parser implementation-specific information can be introduced by similar messages, e.g.

```
parser segment: "Initialization
```

A parser can choose to ignore messages it does not understand.

3.9. Communication with Interpreter

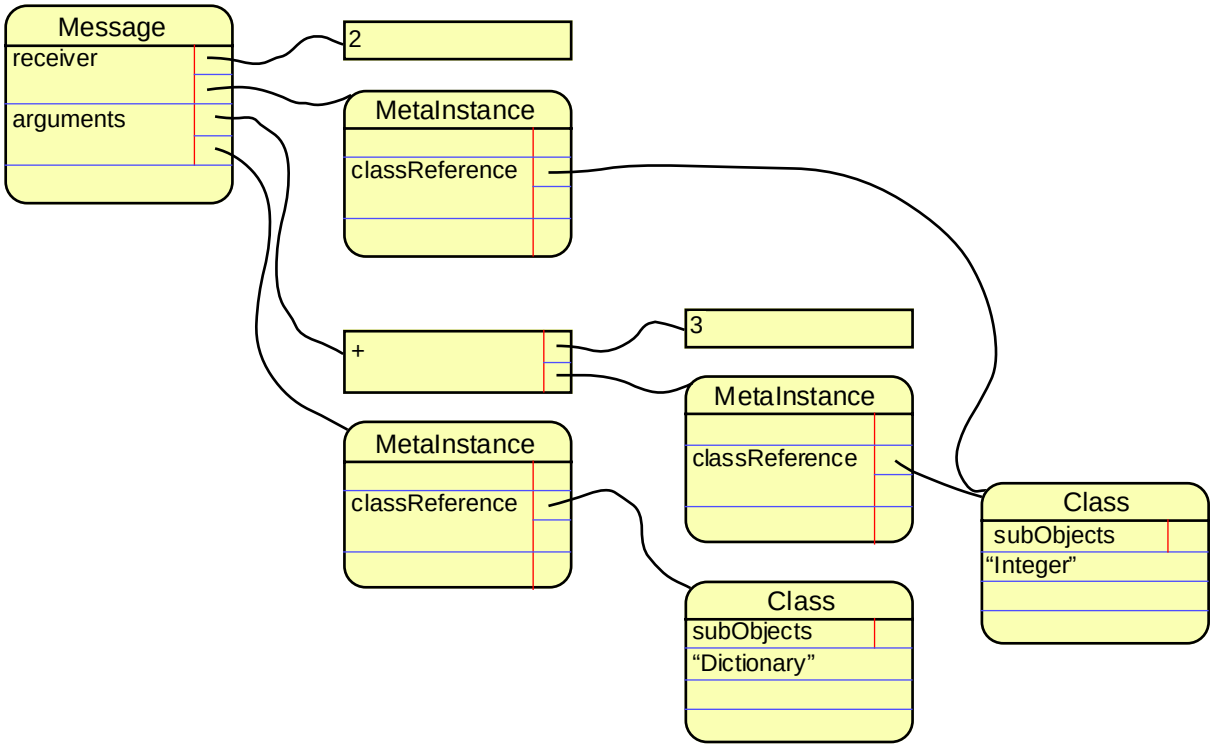
The parser converts the input program into objects, using mostly primitive (i.e., implemented in the host language) object construction and destruction routines. A parser does not do any 'real' processing on the program itself, such as constant expression evaluation.

Figure 2.1 shows how an innocuous-looking expression such as 2 + 3 is translated into objects by the canonical parser (remember that other parsers may have different interpretations for this particular message). While it would be difficult at this stage to explain all its subtleties in complete detail, it is nonetheless worthwhile to take a quick look at it.

The root of the structure is the Message object representing the expression as a whole. The Message starts with a reference to its receiver, and contains further references to a Dictionary of its arguments (a Dictionary is an object that associates names with other objects). Note that references to objects consist of two pointers (pointers are represented by the 'wires' in the figure)— one pointer to the data itself, and another pointer to a MetaInstance. A MetaInstance is an object that describes the structure of a class-instance object. Grunt-92 maintains a strict distinction between an object's data and the data describing that data (called 'meta' data). The Class objects contain code and data common to their instances.

This example has both literal receiver and argument, but the case with variables is exactly the same, except that the objects referred to were not created by the parser but were found by it in one of the scopes. In fact, this is also how the parser found references to the Integer class object (in the system scope).

Figure 2.1
parsed message



4

Object Model

Every artist is a cannibal
Every poet is a thief
All kill their inspiration
and sing about the grief
—*The Fly*, U2 [Achtung Baby]

Objects in modern languages come loaded to the brim with functionality and mechanisms for using them. For example, a Smalltalk object has named and one set of indexed instance variables, is a class instance, and has (fixed) encapsulation characteristics. A C++ object has instance variables, has a class, is never a class, and has a particular encapsulation mechanism. Object Pascal objects are dynamically allocated instances.

These characteristics reflect the various decisions and trade-offs made by the language designers, and are almost always hard-coded into it. In an attempt to provide every user of the language with all the expressive power needed for any specific programming task the language must at once support the complete set of features and mechanisms. This often imposes an additional programming and run-time burden when all the functionality is not actually needed at the same time.

There is hence a trade-off between functionality and efficiency. The trade-off is inherent to the programming problem itself— a persistent class object with run-time method extensions and naming simply requires more processor resources than, say, a stack-based Pascal `RECORD`. A key concept of Grunt is that the programmer, not the language designer, is usually the best person to make the trade-off because he has the best understanding of the requirements of the specific problem. When Grunt is being used with a different parser front-ends for compiling other languages, the 'user' is the designer of the source language, so the decision of what 'an object' is has already been made. It is essential to make as few assumptions as possible.

In Grunt-92, everything is an object. However, instead of loading up every single object with all possible functionality, Grunt-92 concentrates on providing the basic machinery to allow all these different mechanisms to be implemented. The next chapter shows how traditional and some of the more novel object-oriented concepts and mechanisms can be described in terms of it.

This chapter applies the design technique of separation to identify orthogonal (i.e. mutually independent) features and mechanisms that may be found in languages. Because objects are the fundamental concept in Grunt-92, this separation is expressed in terms of objects. Returning to the basics to ask ourselves what really constitutes the barest essence of an object, we find an object in its purest and most denatured form. At the same time the foundation is made for providing the user with mechanisms with which these basic features can be combined, in such a way as is most applicable for the specific programming task at hand.

4.1. Objects

So what is an 'object'? [Micallef88] characterizes an object as a "data structure coupled with a set of operations that describe how the data can be manipulated", and with an explicit interface. The interface consists of messages that can be sent to the object. Wegner's language taxonomy [Wegner87] defines *objects* as having state and an associated set of operations. An *object-based* language is one which supports objects as a language feature. Grunt is object-based by these definitions.

Grunt-92 objects are a slightly more generalized notion: they represent but do not necessarily consist of data, and need not actually be able to respond to messages at all. Object operations, or *methods*, are not so much "coupled" as 'associated' with an object, in that they can but do not need to be defined by the object itself— in particular, the methods of class-instances are not defined by the instances themselves but by the class-object. In the case that a method of an object is defined by the object itself, the operation is part of the data of that object. Indeed, methods are themselves represented by objects.

In any case, the operations that an object can respond to are invoked by the sending of a *message* to the object. The messages that an object responds to, or *understands*, do not necessarily allow manipulation of this data but rather are requests to the object to perform certain services. Carrying out an operation and answering an operation as data are two different things.

Everything is an object in Grunt. Objects usually have a definite internal *structure*, formed by an internal layout of subobjects (sometimes called "fields", or "members"), but this is not strictly necessary— there are structureless objects representing 'raw' data without any 'meaning'. Any structure that an object does possess is not defined by the object itself but by an associated object called its *metaobject*. Certain types of metaobject refer to other objects (e.g., a class-object) to completely define their referents' structure, other types do not. Separating the data itself from its interpretation proves to have a number of advantages.

Also not part of the standard equipment of an object is a name that identifies it. Any name which can be used to access an object is bestowed by specialized dictionary objects. This is analogous to a file on a file system— the file name is not part of the file itself but provided by the file system. The separation makes sense because the object-to-name mapping is not one-to-one: there can be objects without names, or multiple names referring to the same object.

creating objects

Object creation, although often conceptualized as an atomic operation, typically achieves a number of quite distinct aims: storage allocation, construction, and name association. Storage allocation reserves memory for the object to live in; construction builds an object in this storage; and name association provides a means by which users can refer and hence use the object. However, the language designer has tied these operations together in such a way as to give the user the impression of a seamless instantiation process. When an object is destroyed, these actions are reversed: names are disassociated, the object is deconstructed (or, "destructed"), and the storage previously used by the object is deallocated. Each of these separate operations will now be discussed separately.

> It is useful to think about what it means to 'create' objects. The word "creation" implies that something is made out of nothing. It is true that the storage used to hold a new object existed before. When the object is created, this storage may need to be initialized but even this is not always really necessary. In fact, the only new 'thing' that is created is our interpretation of this storage— that is, the metaobject.

allocation

An object occupies storage. Before an object can be constructed, storage must be reserved for it; this is called **allocation**. Allocation does not affect the storage itself; it is merely put aside and made unavailable for other use.

The reverse of this process is **deallocation**. The compiler can and should warn when storage is deallocated that is being used for objects that were neither

explicitly nor implicitly destructed. It should also warn when an object is destructed for which there are still other references in existence.

Different kinds of storage are called storage classes. Storage is managed by instances of Storage.

construction and destruction

Specific methods called **constructors** convert raw structureless storage into proper objects. A constructor is simply a type conversion that converts raw data into an object. There is no special grammatical or functional support. The system inherently forces a constructor to be invoked before storage can be used as an object of the specified type, because otherwise no corresponding metaobject exists (unless, of course, the storage is type-coerced). By convention, constructors have names which begin with new:.

Similarly, an object **destructor** disassembles an object to leave only the raw storage. Destructors' names typically begin with old:.

The notion and terminology of construction/destruction is obviously derived from that in C++. However, because constructors and destructors are ordinary methods, they do not have all the additional restrictions that they do in C++:

- Construction of subobjects is completely under program control. In C++, construction of base classes and members is in declaration order [§12.6.2] and hence impossible to change the manner, order, or necessity of construction at run-time. Object Pascal does have this flexibility, but at the cost that the compiler does not guarantee that all or any object fields have actually been constructed
- Furthermore, bases and members must be constructed in a simple invocation of a constructor, which is executed before the constructor function body. Any actions to be performed prior to the constructor call must therefore be written as `expression`s, which precludes using `statement`s and declarations
- Named constructors/destructors, because in C++ constructors and destructors are specified using special names [§12.1, .4]
- Parameters in destructors, which is not possible in C++ [§12.4]
- Multiple distinct constructors/destructors with identical argument lists, because constructors can only be differentiated based on their argument lists and destructors cannot have arguments

Because objects can define their own behavior without having to rely on classes, a class-object can define any number and kind of constructors specific to its instances without the need of resorting to metaclasses (§3.2), as in Smalltalk [pp. 269-272].

A constructor or destructor that is invoked by a language without an explicit specification from the user is called **implicit**. Constructors and destructors without parameters are natural choices for implicit use, since the compiler does not have to supply any other information to complete the process.

naming

Grunt objects do not have names inherently. Any object-name associations must be made externally to the object itself. The separation of the name from the object itself is quite natural; consider first of all that certain types of objects (such as literal and temporary objects) have no name at all:

```
a + (Integer TEMPORARY new: 2)
```

Here, the literal 2 is unnamed, as is the explicitly instantiated Integer object which is initialized with the value of the literal. On the other hand, some objects have more than one name:

```
Window  DYNAMIC new | aWindow.
aWindow | theSameWindow
```

Here, aWindow and theSameWindow have been made to refer to the same dynamic Window object with the tag operator.

This is akin to references in C++, but note that where Grunt allows the name to be subsequently reassociated with a different object, in C++ is not possible to 'reseat' references.

The Scope system objects support object-name links.

data

A structureless object (**data**) is created by obtaining storage from a storage allocator:

```
static new: 10 | data
```

creates a new data object and metaobject (which is an instance of Meta) and returns a reference to this object. The tag associates the name data to the object. The created object is not an instance of any class, does not have any subobjects, and indeed doesn't have any data at all. Additional structureless data may be added to the object by sending its metaobject the message addData:

```
data` addData: 10
```

adds ten bytes of data to the object. (Since this is 'pure, non-object' data some other object would presumably put it to use for itself, e.g., for some type of buffer.) This and the corresponding message removeData: can be sent repeatedly throughout the lifetime of the object. If it is known that the size of an object will not change in its remaining lifetime, its metaobject can be sent the message constant (cf. §4.7), which prevents further changes to the size or structure of an object:

```
data` constant
```

This is important information, since it may allow the interpreter or code generator to make certain assumptions that may increase the efficiency of using this object, and hence other objects that use it.

objects

Whether one considers the data created above as an object or not, since it does not actually respond to any messages, is a matter of definition. A more conventional kind object that responds to messages and has internal subobject structure can be created from Object:

```
static new: 0 | object.
Object new: object.
```

is an 'singleton object', not an instance of a class, but with a definite internal structure that is currently void. New objects can be added to it:

```
object` add: (static new: (Integer sizeof)) as: "i.
```

This new subobject can be subsequently referenced as anObject [] "i. The subobject can be created in the object itself by first allocating storage in the object:

```
object` new: (Integer sizeof) as: "j.
Integer new: (object [] "j).
```

In fact, any message that returns a result can be used to add new objects:

```
object` add: (2 + 3) as: "sum.
```

Subobjects can be removed from the object with remove:

```
object` remove: "sum.
```

Again, once it is known that the structure of an object will not change, it can be fixed:

```
object` constant.
```

4.3. Metaobjects

A **metaobject** is an object that defines structure on and provides access to another object, called its **referent**. There is a one-to-many correspondence between objects and metaobjects, that is, an object may have any number of metaobjects referencing it. The existence of metaobjects results in a clearer separation of the actual 'entity data', the data that actually represents what the object is modeling; and the 'meta data' that does not actually have anything to do with the entity modeling but is necessary to get the object 'working' correctly.

Sending a message to an object or accessing its internal structure, requires knowledge about the object's structure and hence access to both the object and its metaobject. Since an object does not generally store pointers to its metaobjects, nor is it necessary that a metaobject contain a pointer to its referent, the responsibility of correctly maintaining the association lies outside of the object. This is usually done automatically by the system objects or the parser in the form of *references* (§3.4) so that a programmer normally should not need to worry about maintaining the proper association at all.

Metaobjects are created whenever a reference to an object is created. Since metaobjects are themselves proper objects, a metametaobject is created when a reference to the metaobject is created. This happens in particular when accessing subobjects through the metaobject, for example, in message sending. In principle any level of metaobject can be created, but in practice the 'hard-coding' of primitive code limits the typical use to these two (the metaobject and the metametaobject).

> Metaobjects are really not as radical a concept as it may appear at first— in fact, they are used implicitly by any typed programming language. The difference is that in other programming languages the metaobject is not accessible to the user, but internal to the compiler. Formalizing the metaobject concept allows the programmer to obtain and use this structure information, and permits the introduction of new kinds of data structuring that were not built in to the standard system.

A metaobject responds to most messages (other than constant or volatile, for example, cf. §3.3) by returning a reference to its referent's subobject with the same name as the name of the message— for example,

```
Point new: 2, 3;` x; write                    // writes "2"
```

which creates a new instance of Point with initial value (2, 3), and then writes the value of the subobject with the name x— or

```
Array new; `3                                 // references the third element
```

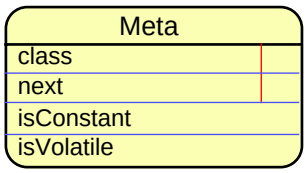
which would create a new instance of Array, then sends the message 3 to its metaobject, causing a reference to the third element in the array to be returned.

> Although metaobjects are limited to being instances, this is not a very serious restriction as inheritance is usually a sufficiently powerful mechanism (although not necessarily the most expressive or efficient). On the other hand, this limitation greatly simplifies the implementation of the compiler— for instance, it means that primitive code always knows where to find a metaobject's class, because Meta's fields are common to each of them. The fact that metametaobjects are always instances allows a primitive implementation of metameta behavior (specifically, the lookup).

meta

Metaobjects are implemented as instances of Meta or one of its subclasses. The way in which different metaobjects provide different types of referent structure is through their different methods of response to messages. Plain Meta instances never return any reference in response to a message, corresponding to the fact that their referents have no identifiable subobjects. Metaobjects of unstructured Data are such instances. Figure 3.1 shows the structure of instances of Meta:

Figure 3.1
meta structure



The class field is a pointer to the class of the metaobject itself, *in casu* Meta. The knowledge that metaobjects are always instances and the availability of the metaobject class means that the MetaInstance metametaobject can be created, and hence that a pointer to a metaobject can always be converted into a reference to the metaobject.

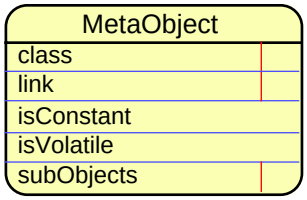
There is a separate metaobject for each reference to an object. Once the last metaobject is gone, the object cannot be referred to by anyone and is therefore no longer accessible. All metaobjects with the same referent are circularly linked through the next pointer. So, an object destructor can tell if it is really the last one to refer to the object by checking whether the metaobject's next points to itself.

The object *qualifiers* isConstant and isVolatile are explained in §3.3.

metaobject

MetaObjects provide *name-associative* (also called "record" or "struct") structure through a dictionary: a readable name is used to identify a particular subobject of the containing object (Figure 3.2):

Figure 3.2
metaobject structure



The first four fields are inherited from Meta. subObjects is a reference to a Dictionary that contains the associations between the names of the subobjects and the subobjects themselves.

Since metaobjects are themselves instances of a class, a metametaobject is always an instance of MetaInstance.

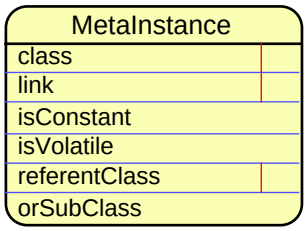
metainstance

MetaInstance provides a similar type of structure to MetaObject, except that the actual structure definition is not contained in a dictionary private to the metaobject itself, but in a class-object that is shared by all instances of the class. This shared definition implies that when the class itself changes, all the MetaInstance interpretations of its instances change as well.

Figure 3.3 shows the structure of MetaInstances. referentClass refers to the referent instance's class. ofSubClass indicates whether the referent may actually be an instance of a *subclass* of referentClass. This usually requires a more costly run-time method-lookup than if the instance class is known exactly, because instance subobjects may be overridden by subclasses.

> Both C's `structs` and `unions` can be represented by instances of MetaInstance. In the case of a `struct` different names are associated with different metaobjects and referents, while a `union` may associate different names with the same referent but different metaobjects.

Figure 3.3 metainstance structure



system overview

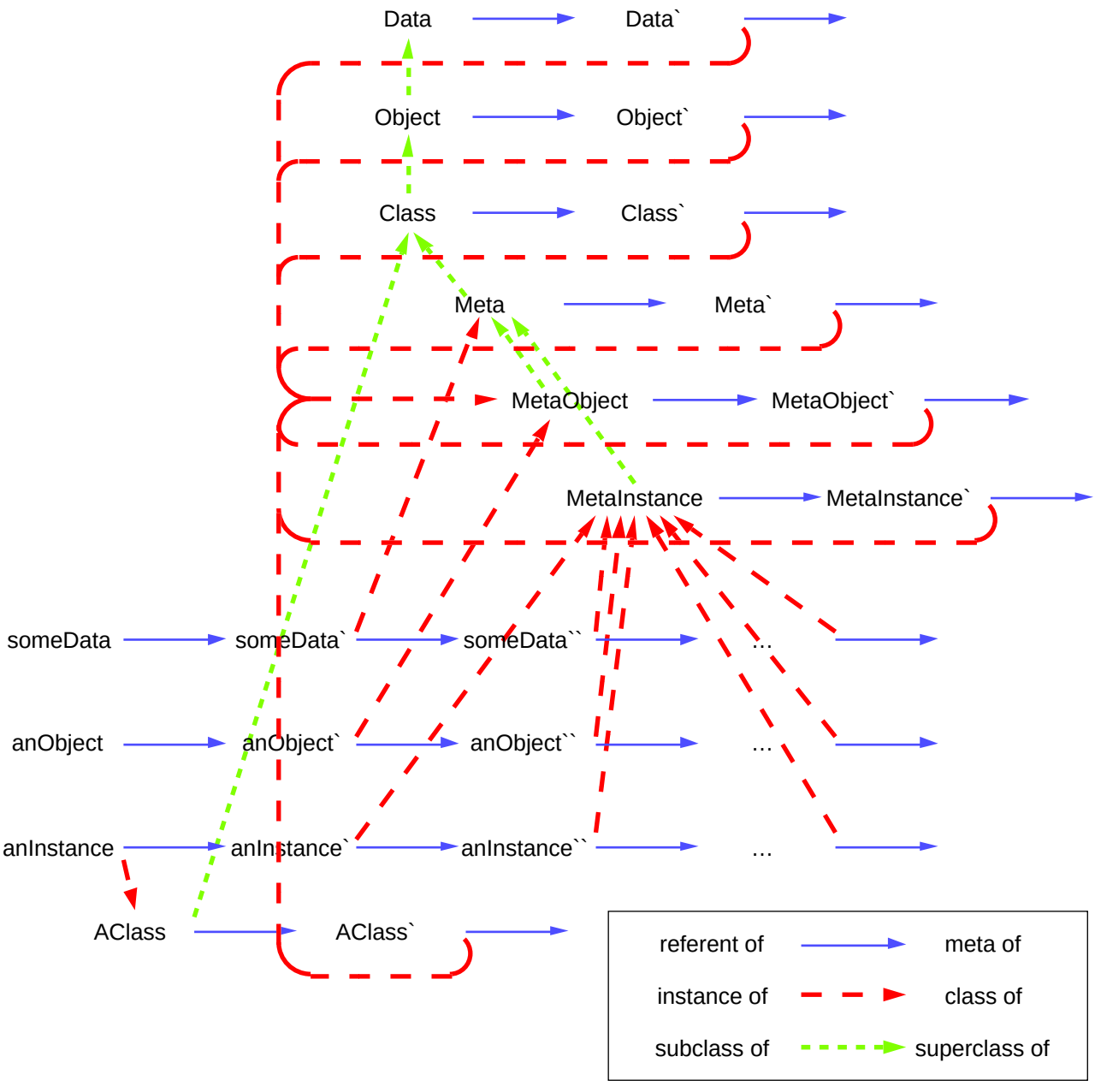
Figure 3.4 gives a diagram of some of the fundamental system objects and how they relate to one another.

First of all, note that the referent objects are shown on the left side with progressively higher meta levels to the right. For instance, near the bottom of the figure, anInstance is the referent of its metaobject anInstance`, which in turn has a metaobject anInstance``, etc. Remember that higher-level metaobjects are created when they are needed, that is, whenever a reference to a referent is created.

Class-subclass and class-instance relationships are shown vertically. MetaObject and MetaInstance are both direct subclasses of Meta. All metaobjects are instances of Meta (such as someData`, meaning that its referent is a simple chunk of data) or one of its subclasses, MetaObject (anObject` and all of the metaclasses), or MetaInstance (e.g., anInstance` and all of the metametaobjects).

Note that classes are objects and not instances of a metaclass, as they are in Smalltalk. The reason that Smalltalk uses metaclasses is that only a class can define behavior for objects (i.e., always instances), and objects (even classes) cannot define their own behavior. This means that class methods, such as methods for creating instances, must be contained in another class. Early versions of Smalltalk [p. 76] had a single metaclass called Class that contained the instantiation methods for *all* classes, but this presents the obvious problem that either the instantiation behavior of all classes be identical or that some naming scheme be used to discriminate the classes' individual instantiators. Smalltalk-80 solves this problem by automatically creating a separate metaclass to hold the class methods. Since Grunt allows an object to define its own behavior, the unnatural definition of a whole new metaclass for the one class-object is unnecessary, as is the unintuitive requirement that methods for instantiating objects belong in the metaclass and not in the class itself.

Figure 3.4
system overview



other metaobjects

Because other kinds of metaobject may be defined by the programmer by subclassing the appropriate Meta class and redefining the way in which messages to the meta object are converted to subobject references, the range of object structure types is similarly flexible.

One other kind of structure that could naturally be provided by MetaArray metaobjects is *index-associative* (array), which could calculate the location of any of its subobjects from the index. A MetaInstanceArray could do the same more efficiently for an array of instances of the same class.

There are probably other more esoteric, as-yet unexplored applications. For example, it is common practice to represent a file on disk by instances of a class, say, File. But actually this is a misrepresentation, because the instances of File are not really files themselves at all, nor do they contain the file's data. Instead, they really represent *access* paths to the files and impose *structure* on them (through a 'byte offset' scheme) and are therefore perfect candidates to be represented by MetaFiles.

4.4. Qualifiers

Object **qualifiers** are properties which affect the manner in which an object's contents as a whole can be accessed. They correspond to and extend ANSI C's *type qualifiers*. Because object qualifiers affect object access they properly belong in the metaobject.

Object qualifiers transfer to any subobjects an object may have. That is, when a metaobject returns a reference to a subobject in response to a message, it constructs the metasubobject with a particular object quality if either the subobject or the referent itself has that quality. For example, if an instance of Point is constant through a particular metaobject, then both subobjects `x` and `y` are also constant when accessed through that metaobject.

constant

An object can be **constant**, meaning that its value can be read but not changed. An object is given the constant quality by sending its metaobject the message constant. Proper use of the constant qualifier can greatly increase efficiency of the program since it allows the interpreter and the code generator to complete the sending of messages at 'compile-time'. Also, a constant method is available for inlining by the interpreter or code generator.

> The existence of the constant qualifier makes it possible for the exact same object mechanisms to be used in defining both compile-time and run-time expressions. In C, for instance, the `const` qualifier on an object does not necessarily mean that expressions involving it will be evaluated at compile-time: this may actually require reverting to other mechanisms, such as `#define`ing the object as a macro or making it an `enum`erator. Unfortunately, these mechanisms place severe restrictions on the type of object on which they can be used. A macro is only actually a run-time saving if the expression is of a type that can be evaluated at run-time (although it is still evaluated every time by the compiler), e.g. `2 + 3`, but not `ln(2)`. An enumeration must always be an integral type.

A side-effect of the compile-time interpretation of expressions involving constant (or any other kind of) objects is that it helps reduce the number of references to that object. In the extreme case, the object is not referred to any more at all and can hence be eliminated completely.

The constant quality transfers to the metaobject as well— in other words, a metametaobject created through a particular metaobject 'inherits' the qualities of that metaobject. This is because a constant object whose structure was variable would not make much sense. Hence, a constant object cannot have its constancy removed through that metaobject, although the object could be accessed through another nonconstant metaobject.

The result of accessing an object that is constant by one metaobject through a nonconstant metaobject is undefined, since the interpreter may have interpreted messages using that object on the assumption of constancy. If the object is subsequently modified through a nonconstant metaobject, the evaluation of

already interpreted messages may be invalid. Under some target architectures constant objects may be placed in read-only or protected memory. Also, a code generator may generate completely different code depending on whether an object (especially a block or block interface) is constant or not, so using that block correctly may depend completely on the quality being consistent.

Most of the Grunt system objects (including the class objects) have the constant qualifier set. This allows the interpreter to make certain assumptions that make the eventually generated code more efficient. In particular, a message lookup can generally be performed by the interpreter only if a metaobject or class-object is constant— if Integer were nonconstant, for example, + could be made to refer to different methods, or could be removed altogether, forcing an explicit Smalltalk-style message lookup to be performed at run-time.

volatile

A **volatile** object in ANSI C is one whose value cannot be relied on to remain constant even though it is not explicitly changed by the program itself. Its primary purpose in C is its use on objects representing memory-mapped I/O, to force the value to be read each time it is used. In Grunt, making an object volatile prevents the interpreter and code generators from evaluating expressions with it and generally prevents 'optimizations' on that object. It is useful for ensuring that certain expressions or methods are processed through the code generator and actually become part of the object program— there is nothing otherwise preventing the interpreter from executing the entire program. An object is made volatile by sending its metaobject the message volatile.

For example, it might be a good idea to make read and write operations volatile, to prevent the interpreter from attempting user input/output at compile-time, but force them to happen at run-time.

There is such a thing as a constant volatile object. Blocks in the canonical language, for example, are such— although their volatility ensures that they are generated into the target code of a compilation, constancy makes that the code generator can generate an efficient copy of the block in the target language. If a block is nonconstant it is generally not possible to transform it into a single corresponding function

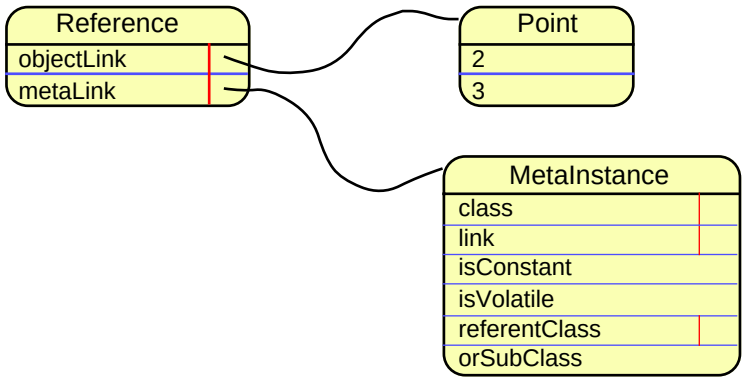
4.5. References

A **reference** represents an 'access path' to an object, which is called its **referent**. It consists of a pointer to the referent and a pointer to the referent's metaobject.

> Although functionally similar, references are not a grammar concept as in C++ where the user cannot, for example, take a pointer to it. The C++ reference functionality is partly provided by the tag operator and partly by the pointer operator.

For example, a reference to a Point object is shown in Figure 3.5. The reference's object link points to the data of the point object itself (the values 2 and 3) while the meta link points to an instance of MetaInstance which identifies the object as an instance of Point.

Figure 3.5
reference example



Because every metaobject either contains a pointer to its metaobject, or contains enough information to allow its construction, the pointer to a metaobject can always be manufactured from a pointer to its referent. After the metaobject pointer has been made, it is conceptually indistinguishable from a pointer to an 'ordinary' (non-meta) object.

An object cannot contain a direct pointer to its metaobject because different metaobjects may be used to access an object. Note that the class field of a Meta is a reference to the metaobject's class, and is used as a source for manufacturing references to the metaobject.

A reference representing an unknown or indeterminate object identity or value has referent as NIL. An object with unknown structure is represented by a reference with a m_referent of NIL.

4.6. Messages

A **message** is a request to an object to perform a certain task. There are three parties involved in the handling of a message: its **creator**, which is the object that created the message; the **sender**, which is the object that initiates the sending of the message; and finally, the **receiver** of the message. The receiver's response to the request is its **method**. A message represents the request, the objects communicated from the sender to the receiver (**arguments**) before the method has executed, and vice versa, those communicated from the receiver back to the sender (**results**) after the method has completed. Finally, a message may allocate storage for objects used only during its execution— these are called **temporary** objects from the sender's point of view, and **automatic** from the receiver's.

In Grunt-92, messages are reified as instances of Message. Objects themselves do not have the capability of responding to messages— this is behavior is provided by Messages and Methods. It is not even necessary for an object to contain its own Methods— an object's meta provides the method lookup and hence its referent's response to messages.

A message consists simply of a Dictionary used to store arguments and results, and a storage allocator for temporaries which are used during execution of the message. The dictionary is also used for associations which are traditionally regarded as 'pseudo-objects'; for instance, every Message must contain an association for the name self, indicating its **receiver**, which is the object for which the message is intended. Other Message types might define other such associations, such as sender, which would refer to the object that is sending the message (which is not necessarily the object that created it).

Note that there is no separate message 'selector'. The method that corresponds to a given message is determined by the receiver's lookup, which can use any or all of the message arguments, as required.

```
Integer new: (automatic new: (Integer sizeof)) | i; := 3.
Message new: (automatic new: (Message sizeof));
        at: "write;
        at: "self put: i;
        deliver.
```

This creates a new Message with the single entry write, and asks the message to deliver itself. The delivery method adds the 'selector' write to the message, followed by the required association for the receiver.

In the above example, it takes a message to create and a message to send a Message. If this were always the case, there would be an unterminated recursion and no work would ever get done. The canonical grammar defines specific rules which allow one to create and send messages implicitly. A synonym for the above is:

```
Integer new: (automatic new: (Integer sizeof)) | i; := 3.
i write.
```

The parser creates a message, determines its receiver and arguments, and adds them to it. Message delivery is a primitive operation in the system. It is possible to override message delivery by subclassing Message, but ultimately the primitive delivery method will have to be called, or a new primitive implemented. In 'interactive mode' (this is every message not inside a block), the parser then delivers the message to the interpreter. While building a block, the message is added to the block rather than delivered immediately.

A default behavior of deliver is to add its associations to the current scope, thereby making the arguments available:

```
scope push: self.                          // message fields
receiver` receive: self.                   // make the object receive us
scope pop.
```

The standard MetaObject response to receive: is to locate the object named by the selector, confirm it is a Block and execute it:

```
(self lookup: name) | method ` isSubclassOf: Method ifTrue: [ method do ]
```

It is the Method's responsibility to perform any required argument type checking.

The manner of the name lookup largely determines the type of code reuse that results. For example, the standard MetaObject lookup uses an internal dictionary subObjects that contains references of the subobject of that object:

```
result = subObjects`(name) @
```

This looks for the object called name in the subObjects dictionary, and returns a pointer to that object in the result reference.

Arguments

Message arguments are added to the dictionary as additional associations. Because associations contain only references to objects, message arguments are always by-reference. By-value arguments must be implemented by the method copying the required argument. Note that because the dictionary is name-associative, the order in which arguments are added to the dictionary is irrelevant. The only case where order matters is when multiple associations with the same name are added. This would only make sense in cases where the arguments are treated equivalently and the order really is unimportant. Consider Smalltalk's explicit messages for creating instances of a Collection:

```
Collection with: anObject.
Collection with: firstObject with: secondObject.
Collection with: firstObject with: secondObject with: thirdObject
```

All three messages obviously achieve the same thing, the only difference being the number of arguments added.

> C provides a similar 'variable argument declaration syntax', but there is never any type checking on the arguments.

Obviously this flexibility requires more work by the compiler. In the absence of explicit information to this effect, the code generator must decide on an argument order itself. It must also decide whether it is possible to send the

message with an unknown number of parameters. If these are undecidable problems in the general case, the user must be asked to provide this information. This is not unreasonable.

> I believe this is a welcome addition. Even though it introduces some problems for efficient code generation, they are not unsolvable.

Arguments are passed by name-object associations in an argument Dictionary within the Message object. In this sense, arguments are always passed by reference. However, a Message also has an allocator from which storage can be obtained for objects whose lifetime is not longer than that of the Message. When the storage for an argument object has been allocated from the Message allocator, the argument effectively has been passed by value.

The order in which arguments are added to the message may or may not be relavant, depending on the responding method.
All listed parameters are added to the message, but the order is not necessarily the same as the order in which they are listed. In fact, in the following example

```
Window newWithTitle: "Hello! at: originPoint size: sizePoint.
Window newWithTitle: "Hello! size: sizePoint at: originPoint
```

both these messages are equivalent, except in the order of evaluation of their arguments, which is always left-to-right.

> Many languages have discrepancies in the way that they associate formal with actual parameters. Many procedural languages are *name-associative* with respect to function arguments in the implementation (that is, formal names may be referred to in the function body) but *ordinal-associative* in their interface (i.e., callers specify arguments by means of their argument list order). Some object-based languages are name-associative in the implementations, and both name- and ordinal-associative in the interface (Smalltalk requires both the argument order and formal name in a message expression). Grunt attempts to uniformize this by providing parameter association by name both for the interface and the implementation.
>
> At least some of this asymmetry may be due to confusion as to what formal arguments exist for. Conceptually, at least, a programmer refers to an argument (or, for that matter, any other variable) by means of its semantics— that is, he usually does not particularly care that it has such and such a name, or occurs in that particular place in a declaration, but rather, that it contains a particular piece of information or serves some certain purpose. Unfortunately, compilers cannot infer such information (nor is it obvious how a programmer would specify it, even if it were available), so a different means of association is necessary. A formal parameter serves as a link for the programmer (which may be either the implementer or the user of a function) between a semantical idea and an actual argument that is passed when the function is invoked.
>
> The question is what is the most useful way of referring to an argument in the absence of such semantical references. Although in some cases the order of an argument may be exactly relevant (e.g. initialization of ordered sets, such as arrays; or non-commutative operators), such ordinal-associative cases are, on average, rare. Programmer-defined names are much more expressive substitutes for semantical references. This makes it all the more surprising that the former method of argument association should be so prevalent. Other *signature-based* languages use an object's signature (the messages that it accepts) as an approximation to its semantics. Although Grunt, in the end, supports any of these types of association (by the redefinition of appropriate methods), a language should, at a minimum, be consistent.
>
> An argument in favor of name-association is the self-documenting quality of messages. This avoids one common (at least in my experience) cause of errors, where the argument order is unwittingly reversed but not detected by the compiler because the types are the same:
>
> ```
> Window::Window(char *title, Point origin, Point size);
> Window window("Hello!", sizePoint, originPoint);
> ```
>
> This demonstrates that type alone is generally an insufficient determinant of semantics.

4.7. Block

In pure object-oriented systems, executable code are objects. Grunt **blocks** are objects consisting of sequences of messages.

Each block has its own scope (the *inner* scope).

The Grunt equivalent of inlining is the interpretation of the block or method within another block. Note that by default, blocks and methods are available for inlining. Making a block volatile prevents it from being used by the interpreter.

Blocks are not class-instances and hence cannot be typed by referring to class names.

metablocks

Note that a block corresponds to a 'procedure' or 'function', in the sense that it is a multiply-usable sequence of operations. It does not in itself, however, provide for an 'interface' on itself. An interface on a block is defined by its metablock.

It is the metaobject's job to create result objects.

The fact that a block always returns a result of a certain type can be expressed by referring to it in the block's metaobject.

5
Mechanisms

In the comfort of this room
the challenge died
—*Horror Head,* Curve [Doppelgänger]

Using the simple object model introduced in the previous chapter, this chapter shows how the traditional object-oriented mechanisms can be implemented in Grunt-92. Although by no means exhaustive, it attempts to demonstrate the power and applicability of the model by describing a range of mechanisms in terms of this model: types, object creation, code reuse mechanisms, and encapsulation.

language shells

In language design there appears to be a certain trend of the identification and separation of extraneous concepts from the proper programming languages. Beginning with Pascal, for example, input/output operations are an integral part of the language itself and fixed in its keywords. In the C languages, however, I/O operations are not part of the language proper but are defined in terms of it, in the form of libraries.

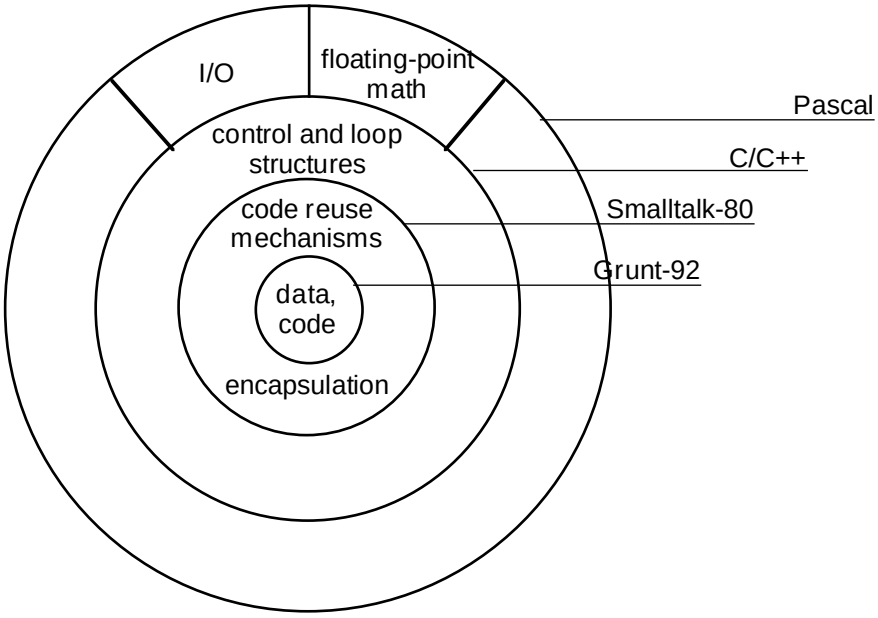
In doing so, the advantages of commonality and portability are not altogether lost because the *standard libraries* are still part of the language *definition.* Since they do take the form of libraries they are orthogonal and hence optional. This has obvious advantages: the baggage of extra functionality need be learned and used only when necessary; also, it can be straightforwardly modified, adapted or extended by programmers after the language has been designed and is used 'in the field'. This flexibility is important, because it is not reasonable to expect language designers to anticipate every possible need and application.

Continuing the trend, a language like Smalltalk even removes such things as control constructs and conditional operations from the language, which are now also defined in terms of the language itself. Although certain fundamental (*primitive*) constructs are predefined by the system, they can be modified or new ones can be defined as required.

Grunt-92 goes a little further in that it also removes most higher-level object mechanisms from the language. This 'stripping away of shells' is depicted in Figure 4.1. Such object mechanisms can be described in terms of more fundamental concepts. Again, the Grunt system predefines certain constructs likely to be of use to a majority of programmers, in the form of *system classes*, it retains the flexibility to modify existing and add new mechanisms. In this way, a programmer is not forced to use mechanisms and features that he does not want.

Grunt operates on the language metalevel itself, such as the object language taxonomy given by [Wegner86]. Note in this respect that the Grunt system class hierarchy mirrors Wegner's application of object techniques to language design—that is, the language metalevel is suffused in the Grunt language. Such taxonomies fail to describe properly Grunt-92, because such a description would fall in the language meta-metalevel.

Figure 4.1
language shells



5.1. Scope

The method of scoping through a dictionary stack is very similar to that of PostScript. Grunt's system corresponds to PostScript's `systemdict`. The next dictionary pushed onto the stack by the parser is `userdict`.

3.6. Storage Classes

«revise» Objects have **storage**. The **storage class** determines 'where' the storage for an object and its metaobject exists. The selection by the programmer of an appropriate storage class for an object is an important consideration which has significant impact on the time and space efficiency of a program.

Storage classes are almost completely parser-specific and are not fixed by the object system. The canonical grammar supports a number of different storage classes, listed here in increasing order of 'expense':

Temporary storage class limits the lifetime of an object to the message in which it is used. Temporary objects are created by the parser, mainly for the storage of results in the evaluation of nested messages and in parenthesized expressions, and are automatically disposed of by the parser after the `cascadedMessage` that uses them has completed. Temporary objects would typically be allocated on a stack.

The system warns about remaining references to temporary objects when they are deleted.

Automatic storage class limits the lifetime of an object to the block in which it is defined. Again, the parser warns about any references that are remaining at

the point of deletion. Such objects are usually allocated within the local stack frame of a block.

Register storage class is exactly like automatic and also limits the lifetime of the storage to the block in which it is defined, but with an additional indication to the code generator that the storage will be heavily used and should be stored in one of the processor's registers if possible.

Static storage class extends the lifetime of the storage to that of the executing program.

External storage class, like static, also indicates storage with lifetime of the executing program but represents storage existing outside of and not provided by the Grunt-92 system itself .

Dynamic storage class is given to objects which have no limitation on the lifetime of an object within the bounds of the executing program. Allocation and deallocation are explicitly specified by the program. Dynamic objects usually exist on a common memory heap.

> There is no exception handling mechanism to deal with things like run-time memory allocation errors.

Indeterminate storage class gives an object life from the point of construction until at least its last use. It coincides with the Smalltalk-80 model of object storage. Because indeterminate storage requires the use of reference-counting and garbage-collection at run-time it is also an expensive form of storage.

Persistent storage class gives an object lifetime beyond the bounds of the currently running program. This probably involves storing the object in some kind of external file on secondary storage. Because secondary storage is usually slow, speed advantages can be obtained by keeping frequently accessed parts of a persistent object, or the object in its entirety, resident in an in-memory storage class object. Note that it pays to have a persistent object in memory when the scope is in use, because in that case pointers to a persistent object can always be resolved to memory addresses.

allocation and construction

It is useful at this point to distinguish between storage allocation and the construction of objects within this storage. The lifetime of the storage logically places limits on the lifetime of the object that uses it, although it is possible to create constructions where storage outlives the object, or the object appears to outlive the storage.

This happens frequently with, for example, automatic objects because in many host systems it is more efficient to create automatic storage for many objects at the same time (when a method is entered) even if the objects themselves are not constructed until later.

> Note also that the term “storage class” is borrowed from C and bears no relation to an object ‘class’ concept. Grunt-92 attempts to improve on it by treating dynamic and other kinds of allocation simply as a storage class, whereas many languages generally use a different syntax and ‘model’ for dynamic objects—C uses `malloc()` and `free()` [B5], C++ the `new` [§5.3.3] and `delete` [§5.3.4] operators, while in Object Pascal all objects are dynamically allocated through `NEW` and in Smalltalk-80 all objects are indeterminate.

The different storage classes are implemented by adding name-reference associations to appropriate Scope objects. For example, automatic storage class is simply the method scope. One way of implementing persistence in a PersistentScope class could be by reimplementing its constructor and destructor methods to read in previous name/object pairs from a disk file when it is instantiated, and writing out all name/reference associations when it is disposed (Chapter 5 describes such a mechanism).

storage qualifiers

In the canonical grammar, the parser decides on a default storage classes for an object, but one may be explicitly specified. Literals and result objects are by default created by the parser with temporary storage class, unless they are named (see below), in which case they are created with automatic storage class.

> 3.6.1. + 3 | i

The literal 2 is created by the parser with temporary storage. The literal 3, since it is named, is created with automatic storage.

Keywords (**storage qualifier**s) explicitly specifying a storage class may be placed before the message whose result it affects:

```
storageClass:
        TEMPORARY
        REGISTER
        AUTOMATIC
        STATIC
        EXTERNAL
        DYNAMIC
        INDETERMINATE
        PERSISTENT
        ε
```

For example,

```
Integer DYNAMIC new: (2 + 3) | sum.
```

creates an Integer instance named sum with dynamic storage class and initializes it with the value of the object with temporary storage class 2 + 3. Both literals are also in temporary storage.

5.2. Typing

type coercion

Metaobjects allow for an elegant expression of the difference between type conversion and coercion. A '**type coercion**', that is, interpreting the same object differently, is achieved by using a different metaobject to access the same object.

Figure 4.2 shows an example of two references with two metaobjects being used to access the same Point. Note that both metaobjects are linked because they have the same referent. Although it is not shown explicitly in the figure, both metaobjects could have different classReferences, meaning that the same object could be interpreted as an instance of a different class. The metaobject could just as easily be any of the other metaobject types— for example, a MetaArray might be used to access the individual bytes of the object.

type conversion

A '**type conversion**', also creates a new reference with a metaobject, on the other hand, creates a new referent as well.

Figure 4.3 shows an example of a Point that is type-converted. The result after the type conversion of the original object is another referent and MetaInstance to access it. This presumably refers to a different referent class, or otherwise the new object is simply a clone. Again, there are no fundamental restrictions on the types of metaobjects that can be returned. A type conversion does not increase the number of references to the original object and hence does not affect interpreter-related optimization issues on that object.

Figure 4.2
type coercion

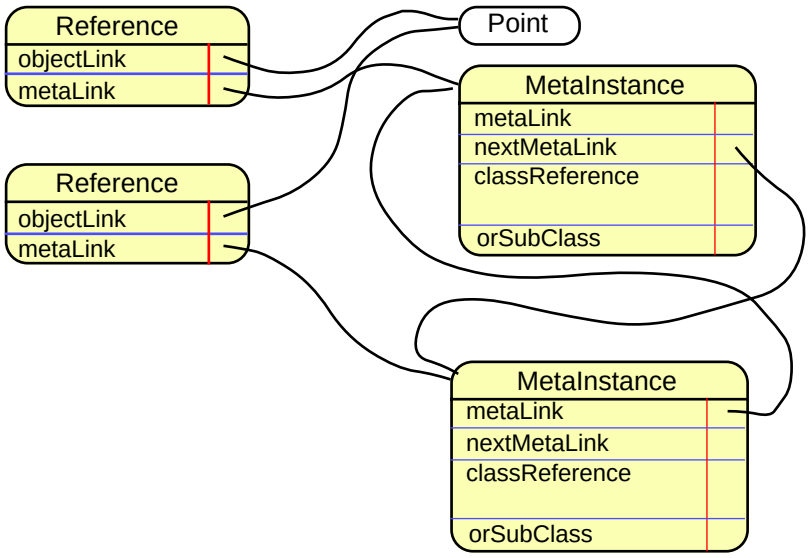
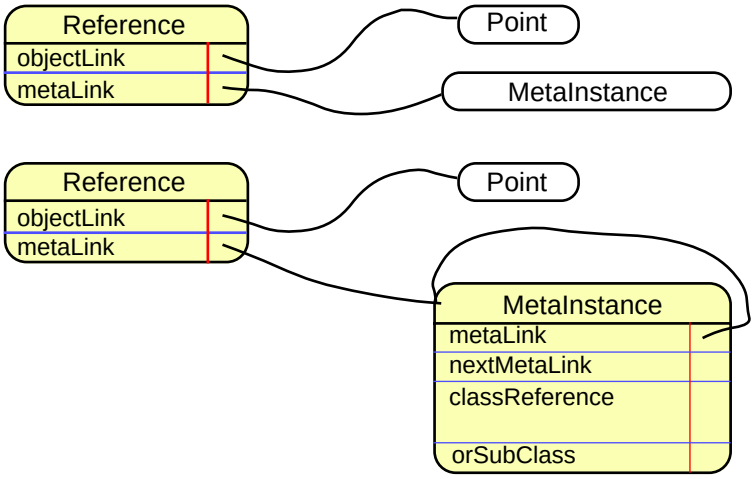


Figure 4.3
type conversion



meta conversions

A special kind of type conversion that changes not only the meta object, but also the *class* of the meta object, is called a **meta conversion**. Meta conversions occur frequently when objcets are being constructed and destructed: they are allocated from storage as Meta objects and converted into exemplars or instances by conversion of their metaobjects into MetaObject or MetaInstance. Sometimes, objects are converted from instances, requiring a conversion of a MetaInstance into a MetaObject.

5.3. Code Reuse

Some proponents of delegation regard [Lieberman86] or even define [Wegner87] delegation as a generalization of inheritance— others still hold that inheritance is more general than delegation [Stein87]. Although we begin the treatment of delegation as a code sharing mechanism before inheritance because it is conceptually simpler, Grunt relies on neither being more fundamental but instead expresses the exact difference between these two code sharing mechanisms in their lookup methods.

There are nonetheless fundamental differences between delegation and inheritance, which is not directly related to lookup. Inheritance has a strict *is-exactly-a* relationship accorded to inherited superclasses, whereas delegation has just an *is-a* relation. That is, with inheritance, it is not possible to use any other than the exact specified classes as superclasses. This is both a strength and a weakness, for it allows more efficient storage allocation (because the sizes of the superclasses are known, or at least, are the same for all instances) at the expense of decreased flexibility in making specific instances.

Another aspect of inheritance is that the inherited member objects are identified by means of their class, and not by a user-specified identifier. This implies that multiple inheritance schemes may have trouble distinguishing between different instances of the inherited class. For example, C++ does not allow the same nonvirtual class to be used as a base more than once in any derivation.

delegation

Note that since this delegation implementation does not require any special cooperation from the 'delegatee', any object can act as a prototype; this includes other objects, instances, classes, and even metaobjects. It is true, however, that in order to make full use of the delegation mechanism, prototypes require the use of the client association in the message, so some amount of foreknowledge is actually necessary. Also note that it is actually just as easy to reuse other kinds of objects than methods— in other words, it is possible to delegate or 'override' data objects.

Delegation has been said to be a design technique suitable for quick prototyping of systems [Lieberman86]. Unfortunately such proponents do not always explain *how* this transition is to be achieved. Here, the transition between a single object (called the **prototype**) to a class of instances is made simply by using the Object's *objectCreator* as a constructor in the Class. The idea is that the same code used to construct the single object can also be used to construct several objects. One other thing is that at this point the programmer does also need to decide if there is any data common to all instances that can be separated and moved to the class object.

inheritance

The object/message paradigm is usually described as objects receiving messages and responding by executing one of their methods. Although this is conceptually correct, certain groups of objects with identical behavior do not of course really have their methods duplicated for every object. Common behavior of objects is stored in a single **class** object that is shared among the **instances** of the class. The second main purpose of a class is the creation of new instances of itself, called **instantiation**. Note that the term "instantiation" properly applies only to objects created by classes.

New classes may be defined by creating new instances of the class Class:

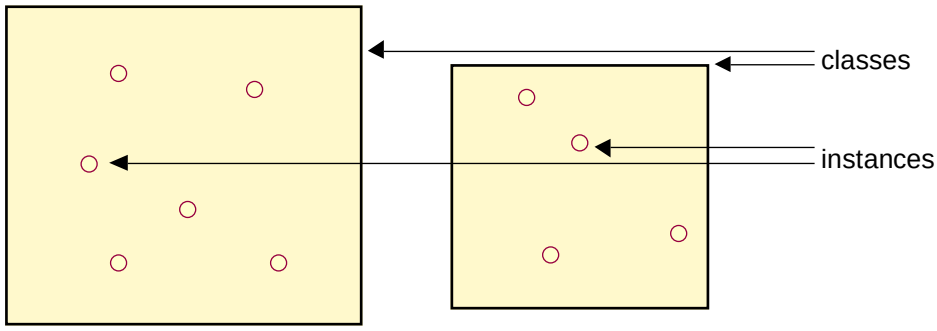
```
Class
        newFrom: superclass
        as: class-creator
```

The *class-creator* is a block that constructs the class-object. This includes the instance methods, and class variables and constants.

Although a class is not itself a metaobject, it can be considered to be part of the 'extended metaidentity' of an instance, since it is actually referred to by instance's metaobjects (instances of MetaInstance).

In Wegner's classification, a *class-based* language is one in which every object has a class. Grunt also supports classes, but because not every object need have

Figure 4.5



a class it is not *class-based*. In fact, by a strict application of Wegner's definitions it is not in fact *object-oriented* because it is not class-based.

> Grunt-92 gives the implementer complete freedom about order of initialization of instances. This is not true in, for instance, C++ where the programmer has to go through severe contortions if he wants to provide a different initialization order in different constructors

Two things can keep Class from figuring out the size of its instances; first, if the number of instance variables added by *instance-creator* cannot be determined, or if the size of one of the subobjects cannot be determined.

Since a class defines structure on its instances, why isn't it a metaobject? In a sense, it is: MetaInstances contain a reference to the class-object. It cannot be an actual subobject of the MetaInstance because there are usually many instances (and hence at least as many MetaInstances) of a class but only one class-object.

overriding methods

A method of a superclass is **overridden** by that of a subclass if it has the same name.

inheritance

An **instance** is an object which has been created by a class-object and hence has certain common characteristics defined by the class. One aspect of instance-objects is that the metaobject defining its structural characteristics is this class-object.

Instances do not really contain the code to respond to messages. In many object languages, then, the message send/receive is skewed, but the asymmetry has been swept under the carpet. Note that there are object languages that allow code to be defined as an instance's subobject but do not allow this code to be invoked as a response to a message.

The class of an instance can be determined by sending its metaobject the message class.

The thing that differentiates an instance from a plain object is the name lookup method in MetaInstance. An instance also has access to names accessible by its class:

```
result := class` instanceObjects`(name) @;        // see if the class defines it
ifNull: [ class ifNotNull:                        // is there a superclass?
        result := class` super lookup: name ].    // check there
```

The first line locates the name in the instanceObjects dictionary. Multiple inheritance uses a similar lookup but looks inside each of several superclasses for the name.

5.4. Encapsulation

The basic object has no built-in access control, although different access control models can be implemented by modifying the message lookup. Smalltalk limits access to data objects to the class itself, and places no limits on code objects. C++ has a three-tier model.

6
External Representation

An intermediate representation (IR) is not strictly required to make a compiler work. [Aho86] gives two reasons for their use: first, a target-independent intermediate representation makes it easier to retarget a compiler to different host systems; and second, that a target-independent code optimizer can be used. This list can be extended with a third purpose, namely, that a good intermediate representation can be used to 'resource' a compiler to different source languages. In fact, this is what has happened with the IR for the C compiler [Feldman79] where the benefits of a general-purpose IR have become clear. An open IR makes the effort of improving and optimizing it more justifiable than would have been the case if it was bound to a single specific source-to-target language translation.

Note that the Grunt system objects serve exactly these three purposes: that is, parsers can be written to generate Grunt objects for different source languages; the interpreter is a target-independent code optimizer because it operates on the objects within the system; and, the compiler can be retargeted to different target languages with new code generators. In other words, we find that the Grunt objects themselves are a natural intermediate representation.

> The Grunt IR is similar to a graph representation: objects can be identified as vertices and references (a dependency on the value of an object) as directed edges. In the same sense, the act of message interpretation can also be framed in more graph-theoretic terms as the elimination of vertices that are disconnected from the main component of the graph. Furthermore, when we identify the 'domain' in which an object exists (the 'source, 'compiler', or 'target' domains) with a vertex color, both parsing and code generation can be viewed as a vertex coloring operation (moving objects from one 'domain' to another) such that remaining message evaluations are monochromatic (occurring within a single domain).

The Grunt system provides support for the storage of objects in a file, called an **external representation** (ER) of the object. By the same token, objects living inside the system are an **internal representation** (although the abbreviation "IR" is still used to mean "intermediate representation"). Using the terminology described in the next section, conversion from an internal to an external representation is called **dumping** and, vice versa, **loading**.

The first two sections of this chapter describe two uses of the external representation, demonstrating how the consistent application of object mechanisms obviates the need for gross hacks to implement new features later on. The last section desribes an implementation of an external representation.

6.1. Load/Dump

There is actually a fourth application of an external object representation. Some compilers provide a 'load/dump' mechanism that can speed up reuse of the results of previous compilations: the user typically specifies a set of files to be compiled, and the compiler 'dumps' a representation of its internal state to a file. In subsequent compilations, this internal state can then be quickly reconstructed simply by reading the dump file back into the compiler (this is not the same as *including* a source file in a compilation, which also results in the reuse of code but requires the source to be parsed anew). The increase in speed results from not having to reparse information in 'interface specifications' each time they are needed.

Unfortunately, because dump file loading is typically tightly bound to a single compiler state, usually no provisions have been made for the loading of more than one dump file in a single compilation. Because the compiler's internal state is not accessible to the user, nor is the dump file format usually documented, it is not possible to exploit the information contained in a dump file (which can be quite substantial in size) in any other way. This seriously limits the usefulness of this technique.

Grunt-92 simply uses the external representation of a parsed program, in the form of an external file representation of the Grunt objects, as its 'dump file'. A modified diagram of compilation using dump files is shown in Figure 5.1.

Because the reloading of a dumped IR is not a compiler 'hack' but cleanly codified in terms of messages to compiler system objects, all the normal object benefits (such as reusability and extendibility) are retained. Moreover, as many dump files can be included as desired, and because reloading a dumped IR is faster than reparsing the source program itself, the IR dump/load mechanism becomes very useful replacement for a straight file "include". The astute reader will already have noticed that the canonical grammar does not provide a strict separation of 'interface' and 'implementation' anyway, so the separation of C-like 'header' files would have been difficult.

6.2. Persistency

An implementation of persistency also requires exactly such an external representation of objects in the system. It was mentioned in Chapter 2 that a persistent storage class can be implemented by saving the objects in a PersistentScope dictionary just prior to it going out of scope. With the ER as described here this can be achieved with the simple act of dumping the dictionary before destructing it. (As will be described below, loading/dumping a dictionary automatically implies the objects associated in it as well.) The objects and names in the dictionary can then be brought back in by loading the dictionary. A PersistentScope constructor takes an external name of the image file (determined by the file system) and reconstructs the dumped objects.

In this light a loaded/dumped compilation state could also be regarded as a collection of persistent objects.

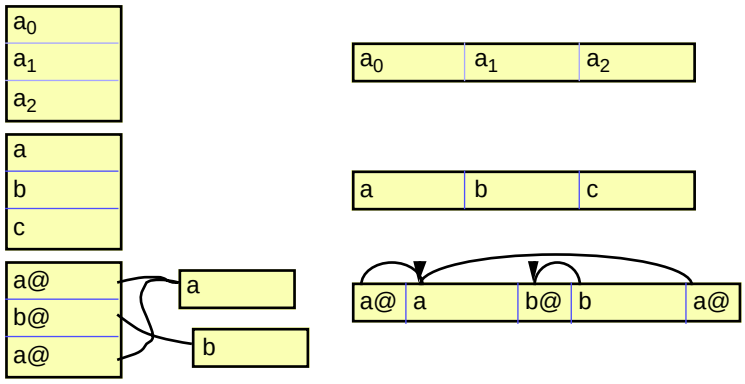
▲ For larger, data base-type operations, the method of loading and dumping the entire dictionary all at once is obviously too crude. First of all, it requires that all the dumped objects fit into memory simultaneously; and second, loading the first object from an image would take an inordinate amount of time. It should not present too much of a problem to implement a modified "SelectivePersistentScope" that is more selective in which objects it loads into memory. At the expense of a higher per-object access overhead, the constructor can simply create the dictionary with the names, and only read in the objects when they are accessed.

> This scheme resembles the Macintosh *resource fork* [Apple85, vol. I], which is a data base of persistent objects called *resources*. The *resource map* is loaded into memory when the fork is opened, and contains enough information for the individual resources to be read into memory as needed. The difference is that the objects must be loaded into memory and changes written back out explicitly.

In principle any object has the capability to be persistent. Some languages require persistent objects to be instances of a specific "`Persistent`" class (e.g., Sina), but this is unnatural because persistency really is only a method of storage and is unrelated to an object's behavior. For example, when a persistent instance is needed it requires the definition of a new class that is a subclass of both the

Figure 5.1 external representation

Figure 5.2
internal and external representations



original class and the persistency class. In a single-inheritance setting such implementations are even more restrictive because either a new class must be defined that is not a subclass of the original class (so its instances have a different type), or the original class must be redefined (making all the instances persistent).

6.3. Implementation

Smalltalk also has an external representation called an *image*, as does APL with its *workspaces*, but only for saving and restoring a single complete object system 'environment' that are not intended to be merged, although utilities are sometimes provided for this purpose. These images are more like the traditional load/dumps in that sense, and therefore suffer from the same problems.

Loading and dumping are mappings from one kind of address space to another: the memory address space of the internal, and the file's 'byte offset' space of the external representation. The only real difference between the two is that the memory is sparse with objects while the file is dense, which is one reason why a complete memory dump is so inefficient.

Grunt ensures that only the actual objects in the memory are dumped by starting with a specified 'root' object and then recursively dumping all objects referred to by the root. When an object refers to another object, the load/dump mapping converts the pointers between the two address spaces and then the objects itself (Figure 5.2). When a referred-to object has already been loaded or dumped before, only the reference needs to be converted. The load/dump mechanisms keep a list of already dumped pointer-byte offset associations for this purpose.

This also solves a problem that occurs with image-based systems that dump entire systems of objects: when one wants to transport a user program from one system to another the entire image must be copied, including many system objects that are not used by the user program at all. The ER includes only the objects actually referenced by the 'root' object.

With the access to type information provided by Grunt it is possible to provide reasonable default loading and dumping behavior for a large class of objects simultaneously, by including information about the object's structure, that is, the metaobject, with the dumped object itself. This is similar to an approach taken by [Abadi89]'s "Dynamic" type, except that it restricts itself to type information only and hence does not benefit from other metainformation such as object qualifiers (constantness of data), or new object structure types.

[Wilson90] describes one implementation of an object *stream* that is based on Object Pascal in the MacApp class library. Because Object Pascal does not provide access to type information, each class must explicitly provide its own

support for dumping and loading. As a program grows to include large numbers of data types, the implementation cost and opportunity for introducing errors increases. Having object images supported automatically is a great benefit and saves the user from the work needed to implement and maintain the individual class load/dump implementations himself, and debug the errors arising from incorrect specification of the actual object structure in the dumped representation.

Since Object Pascal also does not represent classes as objects and has no run-time type information available, it requires an auxiliary user-defined mechanism to identify classes, such as an additional "class ID". The problem is that such class IDs are allocated at compile-time and therefore not constant across different versions of the same program, or between different programs. Since Grunt classes are objects, intance-class links are represented by ordinary references and no other associating mechanism is necessary.

code

Object images are instances of Image, which is a subclass of File. An object can be dumped by sending it in a message dump: to an open image file, e.g.

```
anImage dump: 3
```

Conceptually it might seem better to send a message to the object requesting it to dump itself– however, this will not work for methodless objects that also need to be dumped. The information needed to suspend and reconstruct an object is really dependent on its structure and hence the metaobject. This is reflected in the fact that most objects and classes do not require the rewriting or modification of any dump methods. The dump: method of Image takes care of calling the metaobject dump.

In the simplest case, dumping an object is simply writing its constituent data and metaobject out to a file (metaobjects are simply instances and can be dumped as such). This is the case with plain data objects, i.e., those with Meta metaobjects:

```
image write: (referent@) bytes: (referent` size)
```

File's implementation of write:bytes: causes the referent's data (pointed to by referent@ and with a length of referent` size bytes) to be written to the image file.

Things get more complicated when objects have internal structure. First of all, each of the subobjects needs to be written to the image. For example, for MetaObject's implementation of dump: we do

```
subObjects eachDo: [ image dump: self]
```

eachDo: iterates over the metaobject's subObjects dictionary, and dumps out each of the objects referred to. Something similar happens in the case of MetaInstance, except that it actually defers to a method of the referent's class:

```
referentClass dump: referent in: image
```

which then takes care of dumping the instance's subobjects.

Although subobjects of an object or instance are correctly written out, this is incomplete if the subobject is itself a reference to another object. In the case of references, the object referred to needs to be dumped as well. This can be done by overriding Reference's dump: method:

```
super dump: referent in: image.
image dump: (referent` objectLink)
```

This object may itself may contain references to other objects, et cetera. The recursion ends at the point where either objects are just data without subobjects, or the object has already been written out before: as was already mentioned, the Image needs to keep track of which objects have already been dumped so that they are not written out more than once when there happen to be multiple references to the same object.

7 Interpreter

As stated in the previous chapter, the interpreter can be viewed as a target-independent code optimizer. It is a program that resolves a program as far as possible and outputs whatever cannot be determined with the available information. The output is what is actually used by the code generator. At one extreme it is theoretically possible that a program requires no actual code generation but can be executed entirely by the interpreter.

It is obvious that the reduction of common mechanisms into many fundamental operations makes the execution of even the most simple statement into a non-obvious operation. Yet what a Grunt environment needs to do to execute the statement is in point of fact little different from what the traditional languages do. The difference is that in traditional languages the division of labor is fixed, so that the meta-operations of Grunt are usually all handled by a compiler and the remainder is the compiled program. So while such languages have the division between compile-time and run-time fixed within them, in Grunt-92 this border is fuzzy because traditional compiler tasks are defined in terms of the language itself.

In the absence of restrictions imposed by a source language, some questions may be fundamentally undecidable. This may result in the interpreter having to make 'worst-case' assumptions on the usage of a particular object. This is obviously very undesirable. If the programmer knows that a certain object will only be used in a certain way, but the interpreter does not or cannot infer it from the information available to it, there are two options open: either the interpreter can be modified to specifically detect the special usage, or if this is not possible, the system itself can be modified (and syntactical structures added) whereby the programmer can explicitly request the special usage. It is even possible for the compiler to warn against inefficient usages, and require the programmer to explicitly specify the need for the more general usage. All of this is possible exactly because Grunt is reflective and extendible.

Porting, or 'rehosting', the Grunt-92 compiler requires porting the primitive methods and some bootstrapping code. Once the compiler is running, it can be used to recompile itself with the bootstrapping code replaced with Grunt source.

principles

The parser's part in compiling a program is the conversion of a human-readable form into an equivalent set of objects within the system. The interpreter concerns itself mainly with the message objects in blocks: it tries to send block messages.

Sending a message consists of two separate steps: first, finding the method that responds to that message; and second, executing that method. If the interpreter is able first of all to locate the method it can attempt to interpret it. Since method lookup itself consists of sending a message, to the metaobject, both apparently different steps are in fact very similar.

7.1. Message Lookup

Note that there is a recursion, in that sending a message requires a lookup, which is in turn a message, requiring a lookup, etc.— which must obviously end somewhere: at some point, the interpreter must be able to complete a message lookup by a **primitive** method, i.e., one that is built-in to the host language implementation of the interpreter.

The fact which allows the interpreter to effect a message lookup internally rather than having to send another message is that metametaobjects are always instances of MetaInstance: that is, the object describing the structure of an object is always an instance of some subclass of Meta. This knowledge allows the interpreter first of all to construct the metametaobject and then to substitute a primitive implementation for MetaInstance's lookup. Note that not the metaobject's but the metametaobject's implementation of the subobject lookup is primitive— this permits the future introduction of new kinds of metaobjects as extensions to the system. What the system does *not* allow is the nonprimitive extension of types of metametaobjects, but as noted earlier, this is not a serious limitation.

There are good reasons for wanting to minimize the extent to which the interpreter (or any other part of the system) must be implemented primitively. First of all, any primitive code may be non-portable, although some of this is unavoidable. Second, any non-Grunt code cannot be converted by the code generator, which must therefore also provide its own implementations for all the primitive parts of the system. Other examples of primitive code are the implementations for arithmetic operations and dictionary lookups.

On the other hand, it may be more straightforward to create an initial implementation with a relatively large proportion of primitive implementation.

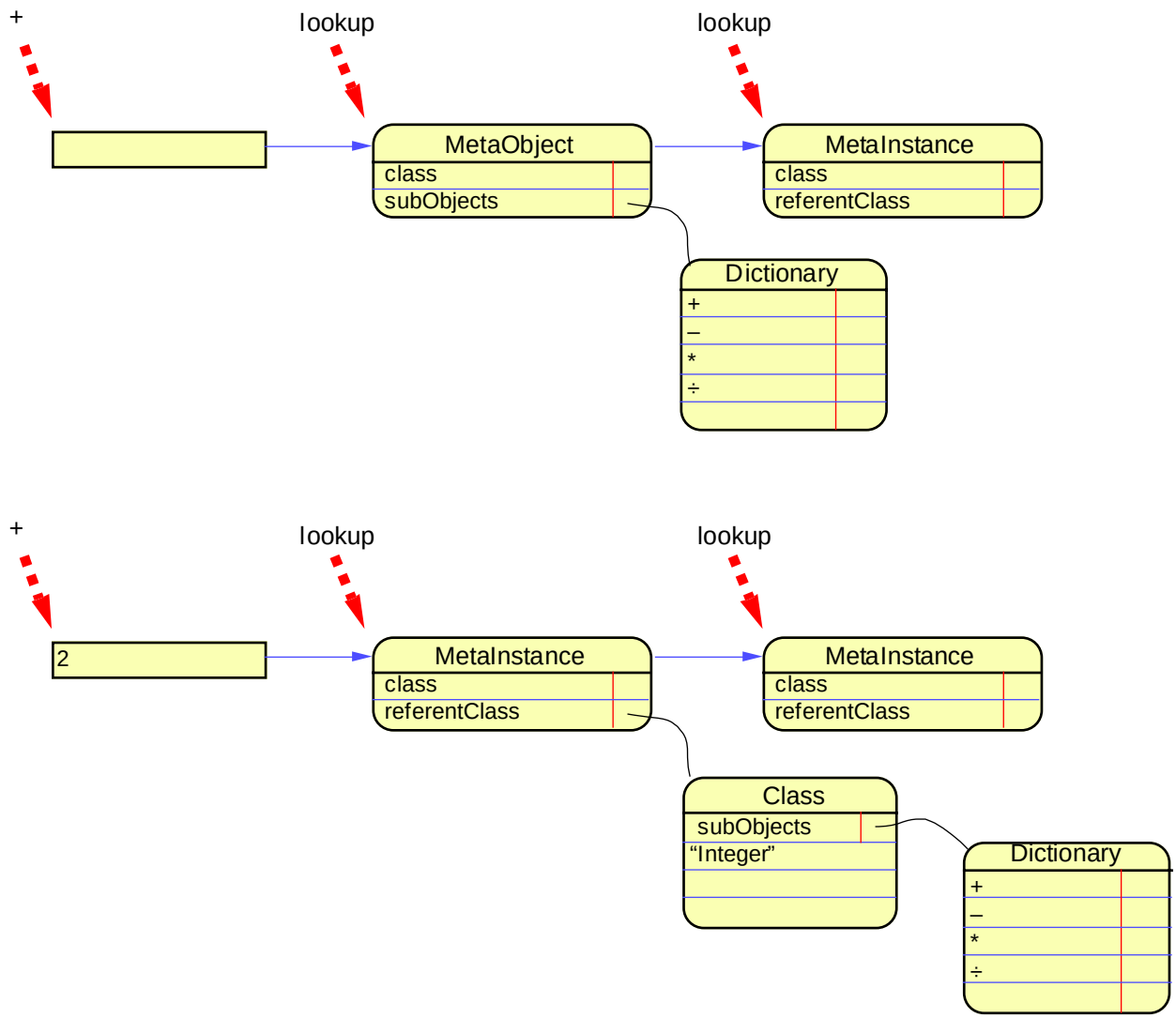
example

Figure 6.1 shows an example of which objects are involved in a message lookup on a plain object, and an instance object 2. Both objects are sent the message +. The interpreter first attempts to find the responding method for +, by sending their respective metaobjects (instance of MetaObject and MetaInstance, respectively) the message lookup "+. Note here that the instance of MetaObject would locate the object associated with + by checking its private subObjects dictionary, while the MetaInstance instance checks through its class. So both messages really do require different lookups.

Sending the lookup message to the metaobject requires a reference to the metaobject, so the interpreter creates a metametaobject. Finding a metaobject's implementation for lookup would then formally be done by sending the metametaobject the message lookup "lookup, but it can be found without a message because this is a primitive operation.

Note that since MetaInstance's lookup is primitive, there might not have been a need to create the metametaobject in the second case (where the receiver was an instance). However, in some cases it may be useful to have a nonprimitive implementation, as explained in the next section.

Figure 6.1
message lookup for object and instance



inlining

The interpreter has another tool available to it in the transformation of blocks, namely **inlining**. Inlining a block means replacing the execution of a block object with a copy of the block itself. Obviously this is only possible if the block is known and only permissible if it is nonvolatile.

There are two main benefits to inlining: first of all, if a block cannot be entirely resolved by the interpreter, inlining that block may still permit a partial in-line resolution to take place, thereby achieving a reduction in executed target code. Second, since inlining reduces the number of references to an object it makes it less likely that it needs to be generated into target code. Although in principle any block of code may be inlined, the benefits need to be weighed against the cost, being greater code size.

In accordance to the two-step process of message sending described above, there are two main types of situation where an interpreter may decide to inline a block of code: the message lookup, and the method itself. Inlining the lookup is

particularly useful, because although in many cases the responding method cannot be determined completely, a significant part of the lookup can still be done at compile-time.

> Note that this generalizes the explicit 'optimization' done by C++ compilers in the virtual member function (C++ does not have virtual data) lookup. In the typical case, a virtual member function access requires an indirection through the virtual member table (*vtable*), but when the class of the receiver is known exactly (i.e., it cannot be a subclass) the lookup is done at compile-time and the generated code contains only a static function call. The generalization lies in the fact that in Grunt this also works for other mechanisms, such as delegation.

For a more detailed discussion of issues related to inlining, see [Hwu89].

example

For example, consider the message a + 2, where a is known to be an instance of Integer or one of its subclasses. To interpret this message, the interpreter begins by attempting to find the implementing method. Following the procedure outlined in the previous section, the interpreter starts with the lookup for + in a's metaobject, creating the metametaobject first (in this example I will show the actual messages in Grunt rather than with the entire pointer structure, which would be quite elaborate):

```
[
        Meta newFromMeta: a` | aMetaMeta.
        Reference newTo: a` through: aMetaMeta | aMetaRef.
        aMetaRef lookup: "+
        ]
```

Meta's newFromMeta: takes a metaobject and returns an approprate metametaobject for it. Reference's newTo: creates a new reference to an object thtough a specified metaobject. Both of these methods are primitives— in fact, the canonical parser automatically creates the metametaobject and the reference whenever the meta operator is used. The third line does the actual lookup of the name + in a's metaobject.

Again, sending a message to the metaobject requires the metametaobject, which we have created. The lookup in the metaobject for lookup: is done by primitive, using the metametaobject to determine that a's metaobject is an instance of MetaInstance. So the message lookup for + that is invoked is the one defined by MetaInstance:

```
[
        result := (referentClass metaFor: message of: referent).
        ]
```

In the first line, MetaInstance's lookup: asks the referent's class (Integer) to create a reference for the subobject message (+) of the referent a, and copies the returned reference in the result reference's metaLink. This involves looking at the class's instanceObjects, where the name isn't found, followed by the class's own subObjects, where it is found.

Although we skip the precise definition of instance-class lookup here, the result is that the metaobject for + in Integer is created, but the part that calculates the address for the method itself cannot, because the referent's class is not in fact exactly known. This part is inlined:

```
[
        result := (referent class @ + (
                referentClass offsetFor: message))
        ]
```

The offset calculation message does not need to know the exact class, and can be interpreted:

```
[
        result := (referent class @ + 18)
        ]
```

for example. Because the referent's class is unknown, this block represents the completely interpreted message lookup for a's +. This lookup is itself inlined instead of the message [a + 2] to become:

```
[
        Message new | message.
        message` receiver := a @.
        message` argument := 2.
        message` method := (a class @ + 18).
        message deliver
        ]
```

The first three messages in this can be interpreted. Leaving just

```
[
        message` method := (a class @ + 18).
        message deliver
        ]
```

for the code generator.

Remember that all of this is actually represented internally in terms of pointers— there is really no string substitution going on here, as the written messages might suggest. This example should further make it clear that primitive methods play an extremely imporant role in message delivery— without them the simplest message would turn into a morass of inlined submessages.

> Although the message delivery probably seems enormously complex and intricate, consider that an ordinary compiler needs to do something quite similar when parsing function calls. The difference is that the behavior of Grunt's internal message delivery is dynamic, depending on specific metaobject's implementations.

remarks

Note that nothing in the message interpretation is geared particularly towards high-speed optimized message lookups. There are no bytecodes, for instance, only full messages. This is in contrast to Smalltalk, where every possible effort is made to make message interpretation as fast as possible, through special caching and hashing schemes, careful selection of primitives that are allotted a bytecode, etc. [Johnson88] [Samples86]. The reason why Grunt-92's interpreter can afford to be more general and slow is that it is not the main engine for program execution— the code generator is expected to generate efficient code.

7.2. Methods

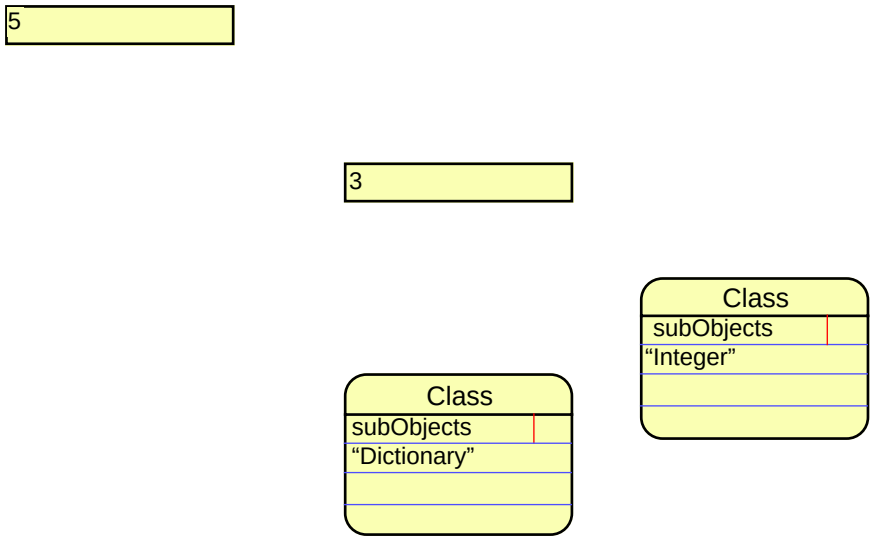
If the approprate method can be found, the interpreter can attempt its interpretation or inlining. Inlined messages are themselves subject to further interpretation. The result is that the original message is removed in its entirety or replaced with a new set of messages. Therefore, the result of the interpretation of a block is in general another block which may in fact be longer or shorter than the original.

example

Figure 6.2 shows what remains after the interpreter has done its work on the example of §2.9 (Figure 2.1). The interpreter (through the metaobject of the receiver) is able to find Integer's + method. This is actually a primitive, so the interpreter can immediately evaluate and return the result 5 in the receiver object. Therefore the interpreter can eliminate its references to the receiver, argument dictionary, and argument objects, and assuming that these were the only references, the objects themselves. The Dictionary and Integer class objects therefore have one less reference.

Note that the complete evaluation of the message does not result in the elimination of the actual receiver and argument object data, only the references. The 3 and 5 are literals, which are created and destroyed by the parser.

Figure 6.2 message after interpretation



7.3. Metaobjects

If the interpreter finds that an object's metaobject was known and nonvolatile at all of its uses, and that no unresolved references to the metaobject exist, the code generator can eliminate the metaobject from the association. This reduction in size may in turn permit the object to be placed in a hardware register at run-time.

In the case of classes, if no MetaInstances of a certain class need to be instantiated at all, and the class is not referred to by anything else (such as a subclass), the class object itself need not be instantiated. This corresponds to the usual case where objects are used without ever referring to their structure. However, when a nonresolvable reference is made to, for example, a metaobject or class, the metaobject or class in question is generated into the output code and so does the corresponding structure information. This would occur when a user explicitly references type information, or if it is needed for run-time type checking.

Again, because structure and type are reified into real objects, they benefit from the exact same interpretation and code generation as ‘ordinary’ objects.

8

Code Generator

Help me find a way from this maze
I can't help myself
But only angels look before they tread
—*Living in Another World*, Talk Talk [Natural History]

A code generator is like a front end, in that it performs a mapping between Grunt and another language— except that obviously the mapping is applied in the opposite direction. Although the mapping may be very simple, a good code generator will take advantage of the particular capabilities provided by the target language. A code generator does not, like the interpreter, have the option of skipping messages and leaving them for another part of the system to resolve.

As stated at the beginning of this report, the first Grunt code generator emits C++ code rather than, for example, an assembly language or machine code. However, much of the discussion in this chapter is meant to be independent of any particular target language and would therefore be of interest to implementers of new code generators. Obvious advantages to generating C++ are that the code generator output is much more readily portable to other platforms as well as easier to read and debug. A disadvantage is that it is much harder to generate 'good' C++ code than, for example, some Assembly language, or even plain C.

This chapter looks at the issues confronting the design of a code generator: the first section shows how the pointer-based object storage scheme of the internal representation can be converted into one that is more efficient and suitable for run-time code. The next section looks into the external linkage of generated code— this is of particular interest for writers of primitive and bootstrapping code, since their routines will have to conform to the linkage requirements of the particular host system.

The C++ code generator is present in the system as the single object named codeGen. codeGen responds to the message generate: by generating the C++ equivalent of the specified grunt object. A simple case is the C++ code for an integer literal, which is the same as the Grunt representation— i.e.,

```
codeGen generate: 1.
```

generates simply "1". The other extreme is the C++ code for the Grunt system itself:

```
codeGen generate: grunt.
```

usually emits C++ code for every object in the system.

All knowledge of C++ and its mapping from Grunt, is contained within this single object— no other parts of the system need to be aware of the range of target languages that they may be translated into. However, implementations that wish to explicitly generate their own C++ code may attach their own code generator to codeGen by calling `codeGenerator`.

8.1. Object Storage

The metaobject concept is an abstraction of referent structure which places no inherent restrictions on the physical layout of objects. In particular, there is no requirement that the object itself be constant, of constant size, or have constant structuring. To support the full generality of changeable object structure, the interpreter's internal object representation is completely dynamic (i.e., pointer-based). No effort is made to place constant or constant-meta objects in special memory that would be 'cheaper' to allocate or access. This would hardly be useful, because it needlessly complicates the design of the interpreter, whereas a speed-up of the interpreter would generally only be of benefit to the program developer and not to its users— they care only about the efficiency of the generated code. Therefore, where a target system provides different kinds of storage which each have their own specific allocation and access characteristics, additional complexity in a oode generator to exploit these different types of storage is appropriate. Of course, a first-order implementation might settle for a simple, purely dynamic storage scheme which mirrors that of the interpreter.

Looking at the specific case of a C++ code generator, a number of storage allocation policies can be identified; some are applicable to other languages as well. When a code generator needs to allocate storage for an object with a constant metaobject, it can put it in special memory that does not support dynamic changes in size. One specific example is the allocation of Method storage (corresponding to automatic storage class): the code generator can allocate storage for all the objects in the dictionary at once at the beginning of a method— when the target language is C this simply means generating declarations of all the automatic objects at the beginning of the function representing the method. Meta-constant objects in static storage can be made C `static`.

When the object itself is also constant, the object can be placed in read-only memory. In C this corresponds to the `const` qualifier.

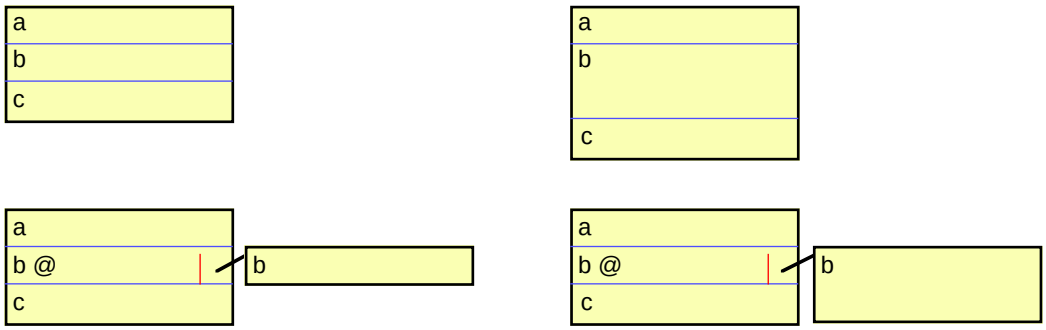
However, when the object is not meta-constant, the object must be placed in dynamic (i.e., resizable) memory because it needs to be able to handle messages requesting changes in size. Moreover, since it is generally not possible to extend the size of existing storage without relocating it in memory, all references to such objects must change. Since it is impractical, if not impossible, to *explicitly* locate every existing reference to or inside any given object, this is best done by relocating the references *implicitly*, that is, by an additional level of indirection.

object layout

A plain data object (i.e. with Meta metaobject) has no internal structure and can therefore hardly be represented other than as a contiguous area of memory. An obvious internal layout of an array of instances (MetaArray metaobject class) is also a contiguous area of memory. There are at least two options for implementing the layout of objects and instances, differing in the way that they handle changes in size of a non meta-constant object.

The first option also uses a contiguous layout of subobjects. However, when the subobject b changes in size, the addresses of further subobjects change too, so address calculations performed with this layout are 'volatile' (they have to be performed every time). The second uses another level of pointer indirection for the variable-length subobjects, achieving a nonvolatile (resolvable) address calculation for all its subobjects at the expense of the overhead additional memory management in object construction and destruction. It should be emphasized that this is purely a trade-off decision regarding the implementation of the metaobject abstraction. Both options are equivalent if all the subobjects have constant size.

compound object storage layout



metaobjects

Normally an object reference consists of a pointer to the object itself and its metaobject. If the metaobject is known and constant at each of its uses, it and does not need to be generated at all.

If the metaobject is still needed, it could be stored just before the object itself. Metaobjects provide a clean solution to the problem of 'where' to put the *vtables* in C++, especially where multiple structured objects are concerned. Most importantly, it provides a solution of *when* to generate metainformation.

One example is in generating code for an expression such as `2 + a`, in which the literal `2` is never referred to anywhere else. `2`'s metaobject is needed only once in the lookup for `+`, but this can be resolved by the interpreter, thus the metaobject `2`` need not actually be generated. The metaobject for `a` in this message is only needed for argument type checking, so if this can be done by the interpreter, `a`` also does not need to be used in this message. This in turn means that the objects and the entire message may be done in hardware registers.

On the other hand, if an object's metaobject cannot be determined exactly, for instance, because it is an argument and the interface did not strictly specify the metaobject, the metaobject is necessary in order to perform or complete the type check or lookup at run-time. So, for example, if the object `2 + a` was passed as an argument to a method which allowed `Integer` or one of its subclasses for that argument, the generated method would most likely require the metaobject for that argument as well.

This means that having primitive objects that are 'real' objects, that is, with all the standard object mechanisms like inheritance, should only actually incur a run-time time or space overhead when strictly necessary. It might be argued that in the example above, generating the metainformation for an `Integer` argument is wasteful, but this is the price that has to paid for allowing subclasses which may potentially override primitive implementations of, say, `+`. If what one really means is "`Integer`, and only `Integer`" this overhead disappears.

8.3. Specific Mappings

The current implementation of the code generator was designed to generate working code for the general case, which is not always necessarily the 'best' C++ code. Although it already does take advantage of some C++-specific language mechanisms, this still needs to be improved. One specific instance of target language mapping is the case of identifier mapping, which is discussed in the next section.

Grunt `Classes` are generated into C++ classes (although this is actually not always possible, cf. §8.6). `Messages` are translated into C++ *expression-*

† At least, this is the case with code generated by AT&T's `cfront`. There is no language requirement that it be so

Table identifier character mappings

*statement*s. In C++ function calls, the receiver ("self" in the message arguments, and "`this`" in the function) and the message selector are implicit and are therefore not passed as arguments.

The code generator always generates name expressions for objects according to C++ scope rules, which will therefore always correctly reference an object whatever scope rules may have been used within Grunt to access the object.

8.4. Identifiers

Most programming languages have a set of reserved words which are not available for use by programs as identifiers. Also, most languages allow only a subset of ASCII characters to be used in identifiers, some are case-insensitive, and often there is a limit, imposed by implementation if not by definition, on the length of identifiers. Code generators need to map identifiers from Grunt to the target language so as to avoid these limitations.

operators

Certain combinations of nonalphabetical characters are valid in C++ as *operators*. Thus, the method name [] is mapped to `operator[]`. Non-method identifiers (such as Integers) are not thusly mapped.

characters

Characters that are not valid in C++ identifiers are replaced with a three-character equivalent that represents that character:

!	_ex	)	_rp	;	_sc		
"	_dq	*	_as	<	_lt		
#	_nm	+	_pl	=	_eq		
$	_dl	,	_co	>	_gt		
%	_pc	-	_mi	?	_qm		
&	_am	.	_pe	@	_at		
'	_sq	/	_sl	_	_us		
(	_lp	:	_cl				

For example, the identifier new: is translated into `new_cl`. Strictly speaking, there is a conflict with identifiers in the same scope that themselves include these equivalents, since they are not changed under this mapping— for example, the Grunt identifier `new_cl` also maps to `new_cl`— but this probability is presumed to be negligible. Also, no mapping is currently implemented for nonprintable or 'high bit set' 8-bit ASCII characters.

length

The standard [ARM §2.3] specifies that C++ identifiers may be arbitrarily long, but accepts that implementations or other parts of system software may impose limits of their own. The code generator implicitly assumes that generated identifiers do not exceed these limits.

reserved words

A list of C++ reserved words is given in [ARM §2.4]. A Grunt identifier which is a C++ reserved word is prefixed with "`k_`" (for 'keyword'). For example, `do` is mapped to `k_do`. The reserved word mapping is done after the character translation.

singleton objects and classes

Since C++ does not directly support the use of classless objects, a single C++ class is created for such objects, which is instantiated only once. There would then be a conflict between the C++ class name and the name of the class-object instance. Also, since Grunt classes are themselves classless objects, there is a further conflict withthe C++ class that defines the class instances.

The name of the singleton object is used as the name of the C++ instance of the class-object class. This is so that any references to this object can be mapped without translation, like references to any other object. For example, the class-object Integer is called `Integer`. The name of the C++ class defining the singleton object is prefixed with "o_", so the object `Integer` is an instance (the only instance) of `class o_Integer`. For class-objects, the C++ class defining the instances of that class is prefixed with "`i_`". So, all Grunt instances of Integer are C++ instances of `class i_Integer`.

Again, there is a conflict when a Grunt identifier in the same scope also uses the prefix (e.g., `k_do` maps to `k_do`).

8.5. Restrictions of the Target Language

Besides the obvious differences in identifier spaces, there are more fundamental differences between Grunt and the target language. The target language may support structures that are at a higher level than Grunt primitives— a good code generator will detect cases when Grunt objects can be mapped to these structures. An example is classes— Grunt Classes can (to a certain extent) be mapped to C++ classes.

Another source of disparities arises when Grunt allows the expression of a concept that is not supported by the target language. There are fundamentally two different ways to cope with such situations:

- simulate the feature by going 'outside' of the language
- fail, while informing the user which particular usage provided the difficulty, and perhaps some advice on how to avoid it

This section highlights some such problems with generating C++ code, while the following section discusses C++-specific features that the code generator knoes about, or could be extended to take advantage of.

constructed objects

Often, the code generator will be required to emit code for an object which has already been constructed by the interpreter, and which may have been changed since its construction. C++ does not support the notion of directly bringing into existance an object with an already specified state— it requires a constructor to be called, and does not know that the has already been constructed by the Grunt system.

In very limited cases, a constructor is not needed. C++ allows objects to be initialized by an *aggregate* (§8.4.1) provided that the object has no constructors, no private or protected members, no base classes, and no virtual functions (and excepting that at least certain implementations of `cfront` 2.1 do not implement aggregates for automatic variables).

The code generator gets around this by fooling the C++ compiler to think that a C++ constructor has already been called for the object. It explicitly allocates the storage for the object, initializes it from its current state, and creates a C++ reference to the 'constructed' object with the appropriate type. For example, an instance of Method might be emitted as:

```
char s_method[sizeof(i_Method)];
i_Method &method = *(i_Method*) s_method;
method.m_preConditions = NULL;
method.m_implementation = NULL;
method.m_postConditions = NULL;
```

The prefix "`s_`" is used consistently to declare an object's storage separately from its true 'type identity'.

Unfortunately, to be able to initialize an object thusly by each of its members requires that they be `public`. See also the next section on one method to get around this problem.

Meta referents are not instances of C++ classes and hence do not suffer from this problem. They are generated as variables of type `char[]` and initialized by their constituent bytes.

class declarations

Although classes are objects in Grunt, they are not in C++ and hence cannot be declared as such. A class that is an automatic a method† cannot be declared inside the corresponding C++ function. The code generator moves all class declarations before and outside the function in which they are defined.

Note that this creates a name conflict when there are two classes with the same name declared in nested scopes. The code generator should encode the name of the function in which the class was declared inside the class name, but doesn't.

external linkage

Once code is generated into the outside of the system, objects can no longer be identified by definite memory addresses. Hence another addressing method is needed to provide external linkage, which is mainly determined by the target system's object code linker. This usually limits the range of available addressing options to alphanumeric strings.

One linkage scheme would be to simply use the message name as the external name. This type of linkage is type-unsafe, meaning that the same message can be invoked with different types and numbers of arguments. When a method's interface block is unknown or volatile, the method can in fact be invoked with unknown arguments so this type of linkage is sufficient. It is also a good starting point for an initial implementation.

In most cases the method pre- and postconditions are known and constant, so that a method has a definite interface. In this case a more type-safe form of linkage is desirable. First of all, if the message has arguments we can form the external name by concatenating the individual argument names in ASCII alphabetical order. Second, for instance arguments we can use a type encoding scheme such as is given in C++ [§7.2c], with the addition that arguments need encoded whether the metaargument is expected together with the argument or not.

> Actually, Stroustrup himself makes some convincing arguments that linkers would benefit from exactly the kind of information from separate compilations that Grunt provides. Obviously if a linker can look at the interfaces directly there is no need for clumsy name encoding schemes. Of course, for a linker to actually use this information would require someone to implement one.

8.6. Current Shortcomings

As stated, the current implementation does not always map Grunt expressions to their appropriate C++ constructs. This section highlights some of these.

function declarations

The current implementation does not yet generate appropriate C++ argument lists. In the general case it isalways possible to pass a message object as the single argument, but this is unsatisfactory C++ code if the message arguments and their number are known at compile-time.

Another point is that if an argument's meta-information is known at compile-time, it does not have to be passed through the C++ function interface. Grunt itself passes object references by means of their metaobjects. For example, a function taking an Integer argument may be declared as

```
void f(i_Integer*);
```

† And note that all the system classes themselves are in static storage, that is, locals of the Grunt system object

rather than

```
void f(i_MetaInstance*);
```

temporaries

The code generator currently generates an explicit (automatic or static) declaration for all nontrivial (that is, non built-in types) temporaries, even if they are referenced only once. C++ allows anonymous objects of arbitrary type to be used inside an expression, if the object can be created with a constructor. For example, the block `[ 1 write. ]` is generated as

```
{
i_MetaInstance m_tmp0;
i_Integer tmp0 = 1;

tmp0.write();
}
```

whereas it could be generated as

```
{
i_MetaInstance m_tmp0;

i_Integer(1).write();
}
```

access control

The current implementation necessarily makes all members public to support its mechanism of creating preconstructed objects. Another method to accomplish this which would obviate the need for system-wide `public` members would be to address the members by a pointer computed from the member's offset in the object, e.g.

```
char s_method[sizeof(i_Method)];
i_Method &method = *(i_Method*) s_method;
(i_MetaInstance*&) (s_method + 0) = NULL;
(i_MetaInstance*&) (s_method + 4) = NULL;
(i_MetaInstance*&) (s_method + 8) = NULL;
```

This effectively and explicitly circumvents C++'s access control mechanisms.

scope resolution

The code generator currently emits incorrect code for *primary-expression*s which reference an object in program or class scope if there is also a synonymous object in an inner scope. These cases should be detected and proper references using the scope resolution operator generated.

8.7. Messages

Because *each* action of a program is expressed in terms of messages, code generation comes down to translating the appropriate messages into the correct output strings. The actions of the code generator, as those of the interpreter, are ultimately determined by its implementation of its primitive operations: because the generated messages are outside the system, a code generator can take advantage of the specific characteristics of the target environment. In fact, just because the objects are not in the system a code generator actually benefits from having many primitive operations, in contrast to the interpreter, which works most effectively using as few primitive methods as possible.

An initial implementation can suffice with a small number of primitives, but a full-fledged, effective code generator would need to exploit as many strengths of the target environment as possible to streamline the generated code and get rid of unnecessary inefficiencies. This not only includes responses to simple arithmetic messages, as in the interpreter, but any higher-level structures that can be executed efficiently by the target environment. For example, if the target language is C, this would include static `structure` and function declarations, and `const` declarations. Because the code generator is an object in a reified

system, it is possible to apply incremental refinement techniques to its implementation.

As stated before, "executing a method" by a code generator means generating output code. The primitive method implementations thus deal mainly with translating messages into text strings and sending them to the output; unlike the interpreter, which dealt only with in-system objects.

inlining

Just like an interpreter, a code generator can decide to inline blocks, although not for exactly the same reasons: the main benefits to a code generator of inlining are the reduction in run-time time overhead of message send (i.e., function call) and, as described above, the elimination of metaobject references. As opposed to the interpreter, a code generator *does* need to worry about message lookup overhead in its generated code.

Again, the decision whether or not to inline a block is a matter of weighing the advantages against the added cost of increased code size. In some cases, however (like the primitive arithmetic messages, as in the following example), the relative benefits of inlining are so obvious that a code generator need not even waste time thinking about it. In this context, see, for example [Hwu89].

examples

As an example of how a C code generator would interpret a message, we take the sample message in Figure 2.1 but modify the argument to be an argument variable (so that the message would actually pass interpretation); 2 + a.

Because the class of the receiver and the message are known, the code generator finds its primitive implementation of the method Integer +. Although it could generate an actual function call to this primitive, it can inline it instead. Suppose that a is an instance of Integer or a subclass. In that case it has its metaobject in this method as well, but since the primitive only needs to access the object's data and not its structure to perform the addition, we have to convert the reference to a simple pointer to a itself. If the storage layout scheme described above is used, this consists of a simple offset into the combined metaobject/object structure.

The primitive implementation consists of generating the text arg_1 `+` arg_2, which in this case would become something like `2 + a.object`.

Continuing the second example from §7.1, we had:

```
[
        message` method := (a class @ + 18).
        message deliver
        ]
```

Most of the message object, including the single argument, had already been constructed by the interpreter. Depending on which storage layout scheme is used, a pointer to the class of a can be found at a fixed offset from the receiver itself, so a class becomes `a.meta.class`. The definition of the appropriate `structs` is what determines a particular implementation's storage layout scheme. After taking the pointer we have `*a.meta.class` and with the addition `*a.meta.class + 18`. Note that these really were four messages in succession even though they were written in a single line. The string expression is stored in message` method.

The message delivery simply makes a call to the function referred to by the message` method, storing the result in message` result, using the message arguments in the order described in §7.2. This results in the code

```
(*a.meta.class + 18)(3)
```

being generated.

constructors

Grunt constructors are methods which convert data into objects. Grunt guarantees that a constructed object really is an object of the appropriate type by executing the constructor method's postconditions. C++ guarantees construction of bases and members with constructors by requiring them to be explicitly named in a constructor initializer [ARM §12.6.2], which executes before and outside the scope of the constructor function body.

Unfortunately, the order of base and member construction is determined by declaration order and cannot be changed, neither on the basis of parameter or specific constructor. Furthermore, the construction of a base or member can only consist of an invocation of its constructor. No statements can be used in the construction of bases and members, and hence no declarations.

Object Pascal does not have this limitation because members are allocated and constructed explicitly in the constructor. The downside is that the language does not guarantee that all or any of the object fields actually have been constructed.

destructors

Grunt destructors are methods which convert objects into data. Since they are simply a particular kind of method, they do not suffer from the same limitations as their C++ counterparts.

8.8. Exceptions

As no `cfront` releases currently implement exception handling, this is done manually through the use of Fault objects and a set of preprocessor macros which create and test for the presence of these objects.

Specifically, the existence of a Fault indicates that there exists an error condition. Fault objects can be instantiated explicitly to signal a failure, or by using one of the `sfault` macros. After calling any 'result-critical' function (ideally, every function), use one of the `check` macros to check for an error and abort the function.

9 Implementation

Look, Ripley— this is a multimillion dollar installation.
He can't make that kind of decision, he's just a grunt
—*Aliens*

In this chapter we look at the implementation aspects of the system as a whole.

9.1. Bootstrapping

When a system of any kind is to be constructed out of nothing, there is always a problem of the orchestration of this process. This process is called **bootstrapping**, by analogy of pulling oneself up by one's bootstraps. The reflectiveness of Grunt makes this more difficult, since almost every object in the system depends on or refers to other objects. The bootstrapping process is divided into three distinct phases:

- *primary boot* is the phase where storage for all the system objects is allocated and Metas for this storage constructed. This guarantees that references to the system objects, if not the objects themselves, are valid
- *secondary boot* invokes the constructors for the system objects. At this point, the extant references actually become useful in that they refer to 'meaningful' objects
- *tertiary boot* creates auxiliary 'user' objects that may use, but are not used by, the system objects. This includes code generators and front ends

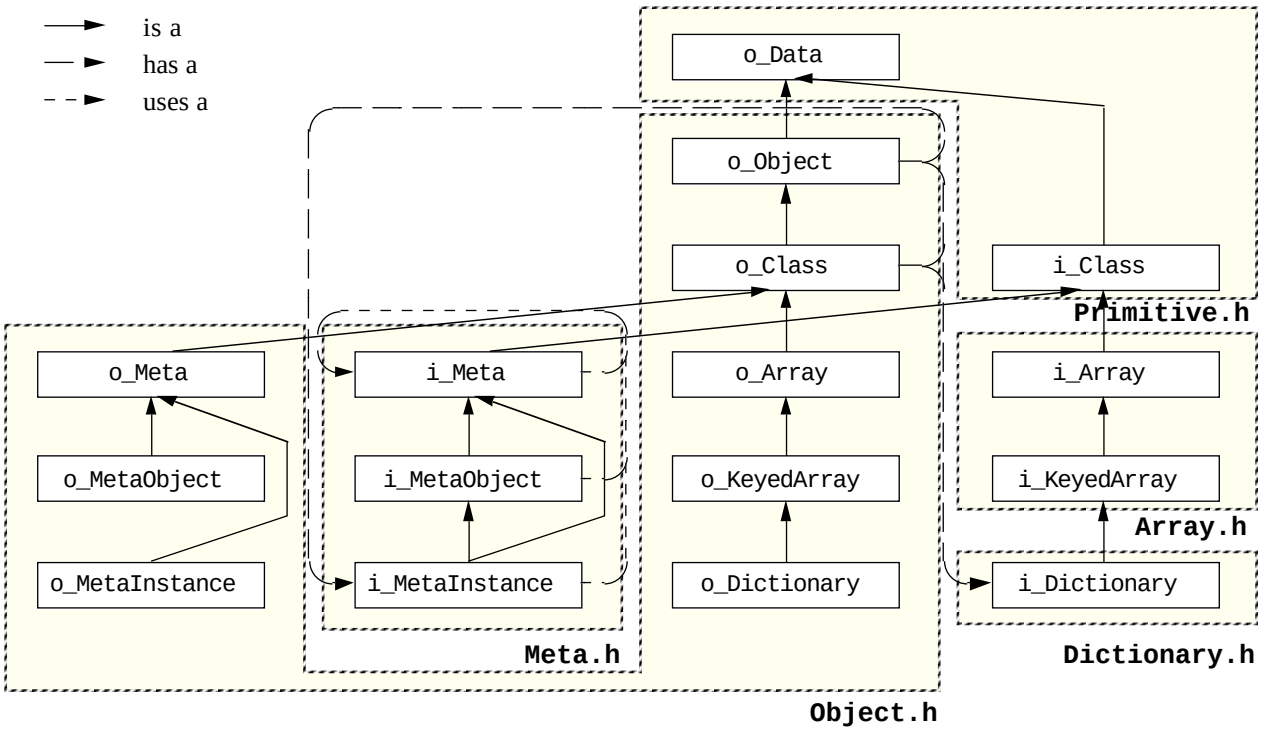
In the C++ implementation, primary and secondary boot phases are performed in the Grunt system object constructor. Since this object is a global static variable, C++ construction rules guarantee that the system is operable when `main()` or anything it calls is invoked[1]. Tertiary boot is performed by `main()` by executing the Grunt code in a file named "`boot.grunt`". This means that extensions to the system can be made by appropriately modifying the code in this file without having to recompile the system.

The Grunt system variable bootLevel indicates the currently completed boot phase— e.g., a boot level of "3" indicates that tertiary boot is done and the system is up and running.

9.2. Source Code

Normally, each Grunt object or class corresponds to its own pair (a "`.h`" header and a "`.cp`" implementation) of source files. At the most primitive levels of the Grunt system the distribution had to be modified to support compilation. The figure shows the dependencies and the distribution over source files of the core primitive C++ implementation classes.

[1] In fact, C++ static initialization rules make an unimplementable promise to initialize a static variable before its first use



9.3 Style Guide

A number of coding styles were adhered to in the implementation, to facilitate reading and understanding of the code. Also, these styles may be used as a canonical example of further development. A summary of these rules is given below:

variable names

Static variables (i.e., globals and static function variables and class members) begin with the character "`g`" or "`G`" (for "global"). Class members' names start with the character "`f`" or "`F`" (for "field"). Constants variables of any kind begin with "`k`" or "`K`". Although strictly speaking not variables, preprocessor macros (such as compilation flags) begin with the character "`q`" or "`Q`". Function variables (automatics) and arguments have no special prefix.

Variables of built-in arithmetic or class types always begin with a lowercase character. Pointers-to-function begin with an uppercase character.

function names

For objects which represent functionality, `operator()` invokes this function. For example, `i_Block::operator()` executes the block of messages.

Objects whose explicit purpose it is to contain other objects may have these subobjects accessed by means of `operator[]`. For example, `i_Dictionary::operator[]` accesses elements in the dictionary, and `i_Meta::operator[]` provides access to a referent's subobjects (the referent is considered to be tightly coupled to its meta).

portability

Although the code was written to be portable across machine architectures, the code in places must make provisions for certain machine-dependencies. Corresponding to the two architectures on which the code has been compiled, the compilation flag `qMacintosh` or `qUnix` will be defined.

10
Conclusions

The words played on around our heads
Perhaps we went too far
We'll soldier on until the end again
—*We Stand Alone*, Ultravox [Rage In Eden]

In this report I have attempted to demonstrate that it is possible to build a reflective object system and language, based on a very small number of elementary principles, that is still powerful enough to support the construction of traditional as well as more advanced mechanisms, while still allowing the generation of efficient code.

The reflectiveness of the system is also the basis for the *extendibility* of its mechanisms. Because the mechanisms are easily modifiable, the language does not have to be 'preloaded' with every possible mechanism in advance, and allows the user to be more selective in choosing the mechanisms most appropriate for a particular task. History has shown countless times that a system designer who hopes to anticipate everything in advance is usually no match for a good user-extendible system.

The most important result of the reification of the compiler, however, is that everything is expressed in terms of and is affected by *exactly the same mechanisms*. That is, parsing, code generation, language mechanisms, compilation, user programs— all can be expressed in terms of the object, metaobject, and message. Once the reification has been done, all the features and mechanisms, all the descriptions in this report apply to everything, regardless. A noticeable result of this is the fact that any properly reified object, including parsers, interpreters or entire compilers, can be used by a program and appear as a part of a generated output program. Also, it is hardly an accident that the same external intermediate representation can also be used as a compilation load/dump mechanism and an alternative for `#include` reparsing, and a persistent storage mechanism— demonstrating that reification can actually result in a code reduction, because of the smaller number of duplicate mechanisms that have to be implemented.

The introduction of both an interpreter and code generators, all acting on the same objects, makes the disctinction between compile-time and run-time mechanisms hazy and unnecessary. Since both are capable of running a program, each can be applied where it is most useful. The reification of the language and compiler means that this does not benefit only the user's program itself. *Any* language mechanism, be it object mechanisms, access control, typing, type information, can be both static and dynamic, because they can all be used and processed by interpreter and code generator alike. A consequence of this is that some of the extreme measures to which compiler implementers have to resort in order to get a dynamic system running as if it were a static system are unnecessary, such as hashed message caches, and bytecodes. Nor does a static system have to pretend to be dynamic with add-on hacks providing quasi-static typing, or run-time type information.

Another, unexpected, observation is that the reflective redefinition of languages can be done in a natural way. Indeed, instead of appearing weak and ineffectual, the simplicity of the reflection allows the user to logically combine principles, and extend them to form better, more powerful, or simply more

appropriate and expressive ones, rather than trying to coerce new mechanisms within the straight-jacket of a non-reflective language. Two examples of this are the more natural solutions of, Smalltalk's 'metaclass problem', and persistent objects.

implementation

The implementation is *not* complete. There currently exists approximately 7,500 lines of C++ code, consisting of a scanner/parser, dictionary/symbol table and list classes, interpreter and C++ code generator, and command-line interface. Much of this is C++ 'bootstrapping code' which could eventually be rewritten in Grunt itself and used to recompile the compiler. I estimate that about 10,000 lines of code are necessary for a basic, first-order version of the system, but much more will be needed to fully incorporate optimization techniques and the Sina front end. An exception is the parser, which is not intended to be rewritten in Grunt.

In the report I have more than once referred to "adding objects to the system". What this really means in practical terms depends on the host system on which, and how, it is implemented. Because there is no standard mechanism in C++ or across operating systems for run-time linking to other code, and the current implementation tries to make as few assumptions as possible about the host environment, it is not possible to portably implement run-time linking. Therefore, adding an object to the system currently implies recompilation and relinking of the system. It is well to realize that this is actually a deficiency of the implementation and not of the design.

future directions

Although it is naturally difficult to predict how well a system will be able to adapt itself to future changes, I think the following examples at least suggest that the current design is robust enough to support some very interesting directions, even though no explicit provision was made for any of them:

The current interpreter uses the simple rule of evaluating every message that it can, while leaving all others for which not enough information is available to the code generator. More sophisticated *message splitting* techniques could be implemented, where certain sequences of messages are interpreted more than once, with more definite information being substituted in some of the cases. The interpretations with the additional information result in more resolutions and hence in generated code that is more efficient. The benefit of message splitting relies on the fact that the more efficient path is taken more often than the general path. This requires some intelligence from the interpreter in deciding which messages are worthwhile splitting, generating the multiple message paths and the additional messages that ensure the right path is taken. Since this kind of replacement and insertion of messages is already a basic task of the interpreter, no fundamental changes should be required.

It is hard to say how well the Grunt intermediate representation would stand up as a basis for 'resourceing' or retargeting to radically different, such as concurrent or distributed, architectures. It is uncertain how well these could be expressed in strictly graphical structures. Moreover, the successful combination of concurrency and code sharing mechanisms remains a non-trivial problem [Matsuoka91]. Note that the volatile qualifier might probably be used to good effect on shared objects, because it forces the interpreter and code generators to explicitly reaccess an object that may be changed.

Recent work [Tanenbaum92] suggests that objects with more flexible access characteristics are very useful in distributed multiprocessor systems. This system uses an *object-based shared memory* scheme, where processors have separate address spaces, but in which shared objects are automatically copied to other processors that access them. Since this involves object access characteristics, this task is cut out perfectly for metaobjects. Once the metaobject was associated to the object, further shared accesses should be completely transparent to the user.

I stated in the report for the Special Topics in Compiler Design course, that I believed that the ultimate test of a language and compiler is an implementation of the compiler in its own language. It seems that the third incarnation of the Grunt language may finally succeed in reaching this goal.

R
References

[Abadi89] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin, *Dynamic Typing in a Statically-Typed Language*, Proc. 16th Symp. ACM PoPL, 1989, p. 213

[Aho86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers—Principles, Techniques and Tools,* Addison-Wesley

[Adobe86] Adobe Systems Incorporated, *POSTSCRIPT Language Reference Manual*, Addison-Wesley, 1986

[Aksit90] M. Aksit and A. Tripathi, *Data Abstraction, Concurrency and Synchronization in Sina*, Memoranda Informatica 90-15, April 1990
Description and examples of the Sina programming language and its concurrency support

[Aksit91a] Mehmet Aksit, Jan Willem Dijkstra and Anand Tripathi, *Atomic Delegation: Object-Oriented Transactions*, IEEE Software, March 1991, p. 84

[Aksit91b] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural, *An Object-Oriented Language-Database Integration Model*, working paper, 1991

[Apple85] Apple Computer, *Inside Macintosh*, Addison-Wesley, 1985

[Ballard86] Mark B. Ballard, David Maier and Allen Wirfs-Brock, *QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods*, Proc. OOPSLA, SIGPLAN Notices, 1986, p. 140

[Björnerstedt88] Anders Björnerstedt and Stefan Britts, *AVANCE: An Object Management System*, Proc. OOPSLA, 1988, p. 206

[Borning82] Alan H. Borning and Daniel H.H. Ingalls, *A Type Declaration and Inference System for Smalltalk*, Proc. 9th annual Symp. ACM PoPL, 1982? p. 133
Describes a separate typing system for Smalltalk, and how types can be inferred in untyped Smalltalk

[Chambers89a] Craig Chambers and David Ungar, *Customization: Optimizing Compiler Technology for SELF: a Dynamically-Typed Object-Oriented Programming Language*, ACM PoPL?, 1989, p. 146

[Chambers89b] Craig Chambers, David Ungar and Elgin Lee, *An Efficient Implementation of SELF: a Dynamically-Typed Object-Oriented Language Based on Prototypes*, Proc. OOPSLA, SIGPLAN Notices, 1989, p. 49

[Coplien92] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley 1992

[Deutsch83] L. Peter Deutsch and Allan M. Schiffman, *Efficient Implementation of the Smalltalk-80 System*, Proc. 11th Symp. ACM PoPL, 1984, p. 297

[Duhl88] Joshua Duhl and Craig Damon, *A Performance Comparison of Object and Relational Databases Using the Sun Benchmark*, Proc. OOPSLA, SIGPLAN Notices, 1988, p. 153
Comparison of relational and object data base models: modeling differences, theoretical and practical performance using a benchmark

[Ellis90] Margaret A. Ellis and Bjarne Stroustroup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990
Definitive reference to the C++ programming language, and valuable implementation hints

[Feldman79] Stuart I. Feldman, *Implementation of a Portable Fortran 77 Compiler Using Modern Tools*, Proc. OOPSLA, SIGPLAN Notices, 1979 (14), p. 98
Describes a compiler implementation, showing the use of Portable C and Standard C intermediate representations and code generators

[Ferber89] Jacques Ferber, *Computational Reflection in Class-Based Object-Oriented Languages*, Proc. OOPSLA, 1989, p. 317
Describes different models of reflection as applied to object-oriented languages, difference between structural and computational reflection

[Frailey79] Dennis J. Frailey, *An Intermediate Language for Source and Target Independent Code Generation*, Proc. OOPSLA, SIGPLAN Notices, 1979 (14), p. 188
Describes an intermediate language, using formalized operator properties, retaining information for code optimization

[Goldberg83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983
The classic and definitive reference work on the Smalltalk-80 language, out of print

[Goldberg89] Adele Goldberg and David Robson, *Smalltalk-80: The Language*, Addison-Wesley, 1989
The language reference part from [Goldberg83]

[Hekster90] Ben Hekster and Peter Middelhoek, *The Grunt Programming Language*, course report, 1990
Report on the original 'Classic' Grunt programming language and its implementation, for the Vertalerbouw course

[Hekster91] Ben Hekster and Peter Middelhoek, *The Grunt-91 Programming Language*, course report, 1991
Report on the Grunt-91 object-oriented programming language and its implementation, for the B.O. Vertalerbouw course

[Hekster92] Ben Hekster, *The Grunt-92 Implementation Manual*, 1992
Detail down to the implementation level of resourcing, rehosting, and retargeting the Grunt system

[Helm90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, Proc. OOPSLA, 1990, p. 169

[Hwu89] Wen-mei W. Hwu and Pohua P. Chang, *Inline Function Expansion for Compiling C Programs*, Proc. 16th Symp. ACM PoPL, 1989, p. 246

[Ingalls88] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph and Ken Doyle, *Fabrik: A Visual Programming Environment*, Proc. OOPSLA, SIGPLAN Notices, 1988, p. 176

[Johnson88] Ralph E. Johnson, Justin O. Graver and Lawrence W. Zurawski, *TS: An Optimizing Compiler for Smalltalk*, Proc. OOPSLA, SIGPLAN Notices, 1988, p. 18

[Kernighan88] Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, 1984

[Kernighan88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd ed., Prentice-Hall, 1988
Definitive reference and tutorial of the ANSI C programming language

[Kim88] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk and Jay Banerjee, *Integrating an Object-Oriented Programming System with a Database System*, Proc. OOPSLA, SIGPLAN Notices, 1988, p. 142

[Koopmans91a] P.S. Koopmans, *The SINA Language: assignment of the Systeem Implementatie Technieken lecture*, 1991
Fundamentals of the syntax and semantics of the SINA programming language

[Koopmans91b] Piet Koopmans, *2. Implementation of Object-Oriented Programming Languages*, 1991
Concise sampling of Smalltalk dialects, particularly regarding their efficiency and applicability to Sina implementations

[Lieberman86] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proc. OOPSLA, 1986, p. 214
Explains the basic differences between sets and prototypes, claims that inheritance is a proper special case of delegation

[Matsuoka91] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa, *Inheritance Anomaly in Object-Oriented Programming Languages*, unpublished draft, April 3, 1991

[Meyer86] Bertrand Meyer, *Genericity versus Inheritance*, Proc. OOPSLA, SIGPLAN Notices 1986, p. 391
Shows some of the possibilities of the different types of genericity, how they can be used in Eiffel and Ada, and how genericity and inheritance can simulate each other

[Micallef88] Josephine Micallef, *Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages*, JOOP, April/May 1988, p. 12

[Samples86] A. Dain Samples, David Ungar and Paul Hilfinger, *SOAR: Smalltalk Without Bytecodes*, Proc. OOPSLA, SIGPLAN Notices, 1986, p. 107

[Stein87] Lynn Andrea Stein, *Delegation Is Inheritance*, Proc. OOPSLA, SIGPLAN Notices 1987, p. 138
Refutation of [Lieberman86]'s claim

[Tanenbaum92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal, *Parallel Programming Using Shared Objects and Broadcasting*, IEEE Computer, August 1992, pp. 10

[Tesler85] Larry Tesler, Apple Computer, *Object Pascal Report*, Structured Language World (9) 3

[Thatte90] Satish R. Thatte, *Quasi-static Typing*, Proc. 17th Symp. ACM PoPL, 1990, p. 367

[Thomas89] Dave Thomas, *What's in an Object?*, BYTE, March 1989, p. 231
Elementary introduction to the object paradigm and its terminology, overview

[Wegner87] Peter Wegner, *Dimensions of Object-Based Language Design*, Proc. OOPSLA, SIGPLAN Notices, 1987, p. 168

Important analysis and overview of the orthogonal 'basis' of the object language design space

[Wegner89] Peter Wegner, *Learning the Language*, BYTE, March 1989, p. 245
Reworked and toned-down version of [Wegner87]

[Wilson90] David A. Wilson, Larry S. Rosenstein and Dan Shafer, *Programming With MacApp*, Addison-Wesley, 1990

Index

Take this kiss upon the brow
And, in parting from me now
this much let me avow:
You are not wrong who deem
that my days have been a dream
—Propaganda, *Dream Within a Dream* [A Secret Wish]
(Edgar Allen Poe)