Contents

* This report was created on the Apple Macintosh Plus, with Microsoft Word and MacDraw II.

Typeset in 10-point New Century Schoolbook, Helvetica and Courier

Preface

Following is the report on the *Grunt* programming language which was written as the assignment for the *Vertalerbouw* course. As per prior agreement, we have chosen to conceive a radically different syntax and semantics, creating an object-oriented language rather than simply implementing the traditional procedure-oriented one. The particular issues and problems which have come up during the development of the language have definitely greatly enhanced our understanding of the semantical and implementational differences between the two categories of languages. Writing the report in English allowed us to use the standard terminology as well as exercise our use of the language, and we hope will not prove too much of an imposition.

All in all, the creation of Grunt, from its design, implementation, testing, debugging, up to the writing of this report spanned more than seven weeks of very intensive effort. Problems with the Computer Center facilities and the departure of one of the authors because of an internship in the U.S. further increased the workload. During the time that both authors were present, there was equal involvement of each in all aspects of the design and implementation of the language—although some of its facets may more pronouncedly carry the stamp of one of the authors in particular, there was always full, active and collective participation. The report was written by Ben Hekster.

The language

The Grunt language is loosely based on and derives its name from the Smalltalk-80 language developed at Xerox's Palo Alto Research Center[1]. Some of the basic concepts of object-oriented language developed here have been incorporated in the language. The name ‘Grunt’, referring to the rudimentary nature of primeval personal communication represents the state of the language relative to Smalltalk.

When designing the language, it soon became clear to us that for various reasons, which will be expanded upon as the particular issues arise, object-oriented systems are most effectively implemented as interpreters. Both Smalltalk-80 and the Macintosh implementation Smalltalk/V are interpreters. Because Grunt, as the product of a course assignment was dictated to be a compiled language, some obvious and also quite a few very subtle implementational and semantical differences between the two types complicated the definition and implementation to the point where some features of the language, notably involving recursion, needed to be somewhat curtailed. As stated, the particular issues will be highlighted in the appropriate sections.

The report

We deemed it appropriate in our case to provide more information than usual on the use of the language—in fact, the entire first part of this report is mainly concerned with the *informal* syntax and semantics of Grunt and covers some of the relevant general truths of object-oriented languages, from a beginning object-oriented programmer's point of view. The second part elaborates on the many and diverse developmental and implementational aspects and difficulties of the language. Please note that it is impossible to highlight *each* decision made in the very lengthy process of designing and implementing a compiler such as this.

The required listings are given in a separate document. They are consecutively numbered and referred to in this report by references such as [L1]. Plain-number references apply to the bibliography in Appendix D.

Observations

We can truthfully state that the course has been of great value to both of us, both in terms of the experience gained in compiler design, working with compiler-generating tools and putting formal computer science and grammar into practice, as well as the insights into the workings and peculiarities of object-oriented languages it has afforded us—more simply, we greatly enjoyed creating our own language. The requirements of this type of language often proved to be such that particular aspects of the generator system, which when being written with conventional procedural languages in mind were taken to be of natural self-evidence, became inconvenient at least. The additional user instructions also added to debugging of the compiler.

We should have paid more attention from the start to the integrity of the procedures. In the case of user errors some actions cannot return a value (e.g., `FindMethod` when no method exists), and return NIL. In the beginning, we had not consistently programmed our actions to be prepared to receive NIL arguments, however. This does occur routinely, because even when an error is detected the compiler is able to continue the parsing process.

Disk quota and budget constraints became increasingly worrying as we neared completion. Regularly, compilation or linking was aborted because the overdraft had been exceeded. We have consequently become particularly adept at typing the following commands:

```
PURGE *.*
DELETE *.LIS;*,*.LST;*
```

A quick mental calculation showed us that using a modem during the weekends, compilation was 90% cheaper, and proved to be quite a bit faster as well. A source-code control utility (such as UNIX's `make`) also would have saved us some perplexed debugging as well as several unnecessary compilations.

But by far the most entertainment was provided by the staff of the Computer Center in the three separate system shutdowns, the accidental termination of our account *after* the compiler was finished, its restoration with week-old files, which a visit to said staff was able to improve to just day-old versions, the discovery that the system no longer recognized us as the owner of our files, which situation, fortunately, a further two visits to the Computer Center was able to correct.

Nevertheless, both of us are greatly looking forward to the succeeding course.

Ben Hekster
Tankelanden 5
7542 DR Enschede
053-764091

Peter F.A. Middelhoek
Glanestraat 19
7555 KW Hengelo
074-911674

1 The Grunt Language

Objects and Messages

In an object-oriented system, an *object* consists of some private data and a set of operations which can be performed on it. Examples of objects could be integer values, display windows and files. A *message* may be sent to an object, requesting that one of these operations be carried out on it. The object's implementation of the reaction to the message is called the *method.* The object to which the message is directed is called the *receiver* of the message. For example, in Grunt, an integer object 3 can be sent a message requesting that the value of another integer object a be added to it:

```
3 + a
```

In this case, 3 is the receiver of the message + a. The integer object a is the *argument* of the message. Methods answer to messages by returning a *result object.* For the + message the result object is another integer object.

Another example is the message write, which requests that the integer object display its value on the screen:

```
3 write                " the screen will display '3' "
```

write takes no arguments. Text between double quotes is called a *comment* and is ignored by the Grunt compiler.

If a message is sent to an object which it does not implement in one of its methods, it is said that the object does *not understand* the message. For example, if we were to ask the integer to compute its Boolean inversion, the system would produce an error message:

```
3 not
?? 1. EUSR : Message not understood in/near line    …
```

An object whose value is explicitly stated is called a *literal*—for example, '3' is an integer literal. 'a' is the name of a *variable* object with an as yet unspecified value. Variable names may begin with any letter or symbol except a double quote '"', parentheses '(' or ')', or dollar sign '$'. The '$' symbol is used to signify character literals, as in a†. Carriage return and tab characters can be specified by $$n$ and $$t$, respectively. There are also two Boolean literals true and false. The complete Grunt grammar may be found in Appendix B.

Objects usually understand an *assignment* message ':=' which can be used to assign values to variable objects, such as in

```
a := 2
```

As literal objects are objects in their own right, they can also respond to messages and their values can be changed, for example, by sending assignment messages:

```
3 := 2
```

Even though literal objects only exist within the expression, this feature can be very useful. To input an object directly into an expression without having to declare a variable, literal receivers can be used:

† It was originally intended that character literals in Grunt have the same syntax as in Smalltalk, e.g., $a. However, the scanner generator requires character literals to be terminated at both ends

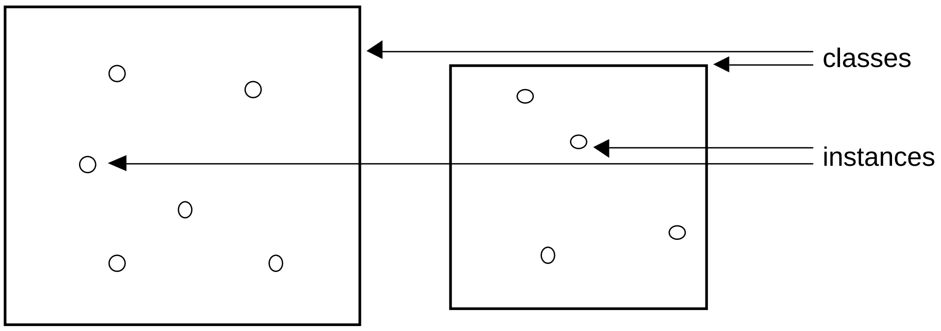
```
(0 read) write
```

The Integer object 0 is sent the message read, which allows the user to enter a value from the keyboard. The resulting object is then written to the terminal.

Classes

Same kinds of objects are grouped into *classes*. There is, for example, an Integer class. Individual objects belonging to a class are called *instances* of the class. A new copy of the private data is created for each instance. For example, 3 and a are instances of the class Integer. The private data associated with classes are called *instance variables,* because they exist independently in each individual instance. In the Integer class, the value of the integer object is stored in its only instance variable.

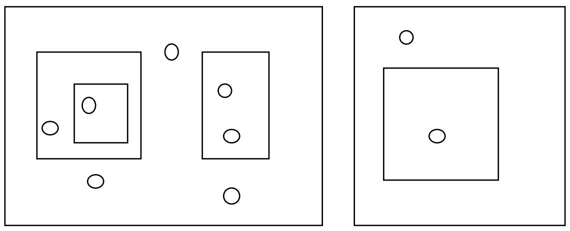
Figure 1.1



Superclasses

New classes can be defined in terms of existing classes by specifying additional private data and operations in the new class. The new class is called a *subclass* of the existing one, which is called the *superclass.* This type of inheritance may be depicted as follows:

Figure 1.2

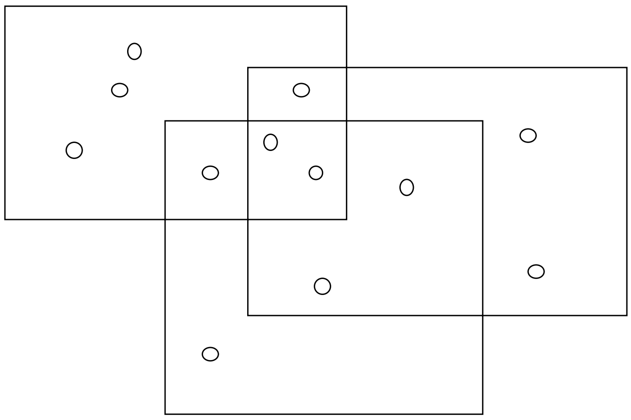


A class may have any number of direct subclasses but only one direct superclass. There is only one class, Object, which has no superclass. All classes are either direct or indirect subclasses of this class.

The subclass *inherits* all of the instance variables and methods of its superclass—it builds on and implements new functionality absent in the superclass. In this way, when a message is sent to a class which it does not explicitly implement, it is passed to its superclass which perhaps does, and so forth, until eventually a superclass is reached which does understand it. In interpreted systems, this actually does represent the manner in which a message is sent—in the Grunt system, however, a check is made beforehand whether a class or one of its superclasses understands the message, which is then sent directly to the first implementing class.

Another form of subclassing, called *multiple inheritance,* was also considered during the design phase but later abandoned due to implementational difficulties. In this scheme, objects can belong to more than one class at the same time. For instance, a new class with both Boolean and Integer instance variables would respond to unknown messages by sending *each* of its instance variables the message in question. In this way, the objects' individual classes decide how to implement the message.

Figure 1.3



Class declaration

A new class Point, consisting of two integer components representing the *x*- and *y*-coordinates of the point, may be declared by the following:

```
"
  Class declaration
"

CLASS Point PARENT Integer       " Inherits its x-coordinate from Integer "
| Integer y |                    " but adds the y instance variable "
" method declarations "
ENDCLASS
```

This declares a subclass Point of Integer with an additional integer instance variable y. Class names must be unique. It is customary for class names to begin with an uppercase, and for any kind of variable to begin with a lowercase character.

The *x*-coordinate of the point does not have to be declared as an additional instance variable because one is already inherited from the superclass. Of course it would have been equally valid to have declared the instance variable x and to have considered the inherited object to represent the *y*-coordinate. It is illegal to declare instance variables of the class which is being defined—this is called *instance recursion* in Grunt, and results in an error.

As it is, Point implements no methods, and any messages sent to it will be deferred to its superclass Integer. For example, if location is a Point, the result of

```
location + 2
```

is the Integer value of the *x*-coordinate of location plus two.

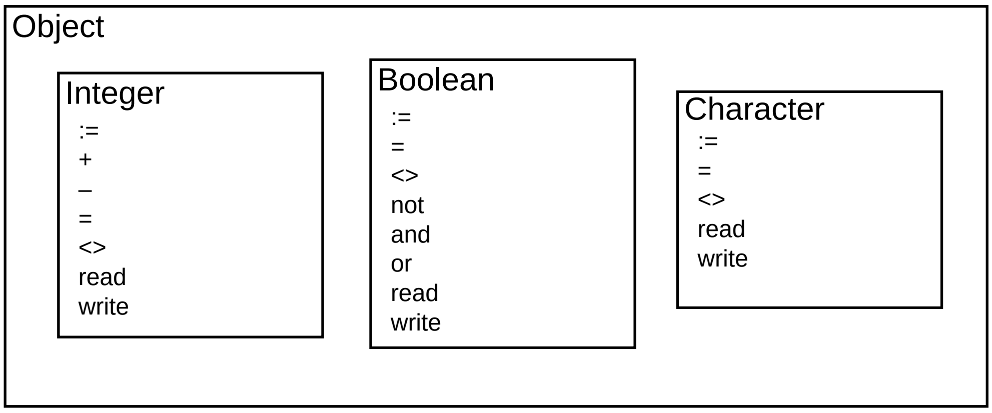
The explicit specification of the superclass “PARENT Integer” may be omitted, in which case the superclass is taken to be the class of the surrounding declaration—new classes may be declared following the method declarations and before ENDCLASS. The source file may itself be considered to be similarly inserted in the Object class declaration (see below) so that outermost declarations will have Object as their superclass if they have no PARENT specification. This form of class declaration may be preferable in some instances where it more clearly indicates the class hierarchy.

A small concession had to be made in giving the Grunt programmer maximum possible freedom in class naming, in order to make the system operable. In the interactive graphical environment of Smalltalk the operator uses his mouse to select expressions and thus send messages to arbitrary objects. Grunt, however recognizes a class Program and begins execution by executing its first method. Programs without this class cannot run.

System classes

Because new classes must always succeed from superclasses, it follows that there must already be at least one class from which these can be defined. These classes must also have some predefined methods, because new methods are described in terms of other methods. The Grunt system provides four such *system classes,* Object, Integer, Boolean and Character. The Object class has no instance variables and no methods—it serves merely as the superclass to the Integer, Boolean and Character classes. The system class hierarchy is illustrated in the following figure:

Figure 1.4



Within the class boxes the messages have been listed which are understood by the class—some of these have already been introduced. Many of the messages are understood by more than one of the system classes, such as := (assignment), = (equality), <> (inequality), read (input) and write (output).

The methods predefined with the system classes are shown below in pseudo-Grunt format. (Method declaration is explained in the following section.) The primitive methods are the only ones which cannot actually be expressed in Grunt itself but have been implemented directly in the native processor's machine language, in this case the ‘targetcode’ language. The declarations are useful, however, because they define the manner in which the primitive methods are to be used (cf. also the next section).

```
"
  System class 'declarations'

  Imaginary declarations illustrate primitive methods
"
" The declarations make use of a nonexistent class Word "

CLASS Object

  CLASS Integer
    | Word i |
    METHOD Integer := Integer value || i := value ENDMETHOD
    METHOD Integer + Integer addend || i := i + addend ENDMETHOD
    METHOD Integer – Integer subend || i := i – addend ENDMETHOD
    METHOD Boolean = Integer comp || i == comp ENDMETHOD
    METHOD Boolean <> Integer comp || i <> comp ENDMETHOD
    METHOD Integer read || i read ENDMETHOD
    METHOD Integer write || i write ENDMETHOD
    ENDCLASS

  CLASS Boolean
    | Word b |
    METHOD Boolean := Boolean value || b := value ENDMETHOD
    METHOD Boolean = Boolean comp || b = comp ENDMETHOD
    METHOD Boolean <> Boolean comp || b <> comp ENDMETHOD
    METHOD Boolean not || b := b not ENDMETHOD
    METHOD Boolean and Boolean op || b := b and op ENDMETHOD
    METHOD Boolean or Boolean op || b := b or op ENDMETHOD
    METHOD Boolean read || b read ENDMETHOD
    METHOD Boolean write || b write ENDMETHOD
    ENDCLASS

  CLASS Character
    | Word c |
    METHOD Character := Character value || c := value ENDMETHOD
    METHOD Boolean = Character comp || c = comp ENDMETHOD
    METHOD Boolean <> Character comp || c <> comp ENDMETHOD
    METHOD Character read || c read ENDMETHOD
    METHOD Character write || c write ENDMETHOD
    ENDCLASS

" The user source file may be syntactically considered to be inserted here "

ENDCLASS
```

Methods

Methods are a class's way of responding to messages which it receives. They can be added to new, but not to existing classes. The following sections explain *how* a message response is effected, and how new methods can be created, possibly altering the operation of existing methods.

Expressions

The result object returned by objects can be used as a receiver or argument in another message, so forming *expressions*. For example, the Integer object 1 can be sent the message – 3 + a:

```
1 – 3 + a
```

which in turn returns an Integer result with value –4 (assuming the previous assignment a := 2 was carried out). Messages are *right-associative*—i.e., the above expression is evaluated as 1 – (3 + a) and not as (1 – 3) + a. Of course, parentheses may be used to alter the order of evaluation.

Multiple independent expressions can be grouped together into a single expression by use of the *dot operator*. Pairs of expressions separated by a '.' are evaluated left to right, and the class and value of the expression are the class and value of the right subexpression. For example, the expression

```
1 – 3 + a . true not
```

returns a Boolean object with value false.

Receiver objects

When a message is sent to an object, the class's method determines what the object's response will be, and consequently, whether the value of the object itself is changed. For example, each of the system classes' interpretation of the assignment message := is to equate the value of the receiver to that of the argument:

```
b := true
```

The assignment messages are actually the *only* primitive methods which do so. None of the other primitive methods alter the value of the receiver, so in the following expression the value of b remains unchanged:

```
b not
```

What messages do, is return a *new* object—in this case, one representing the Boolean inversion of b. Carrying the principle through, expressions represent *new* objects and *not* the receiver of the first expression. For example,

```
(b not) := true
```

assigns the value true to the result object b not, not to b itself. Even the assignment methods return an object, representing the newly assigned value of the receiver. Objects such as (b not) are called *temporary objects,* because they exist because of and only during evaluation of the expression. When forming expressions it is important to realize whether the receiver may be altered.

Method declaration

The newly defined Point class now includes the necessary data, but the additional data is useless because no methods have been implemented to operate on it. We may first of all wish to define a method which can be used to reset the coordinates of the point to zero:

```
"   Point reset
    Resets both coordinates of the Point
"
METHOD Point reset ||
  super := 0.
  y := 0.
  self
  ENDMETHOD
```

Method definitions are given inside the corresponding class definition. (A complete Point class is given in Appendix A.) The above defines a new method reset with a Point result. By convention alphabetic message names start with a lowercase letter. Alphabetic messages which take arguments usually have a colon ':' at their ends.

A method may have *temporary variables* which are created each time it is invoked and are destroyed when it has concluded. As with instance variables in class definitions, temporary variable declarations are placed within vertical bars '|'. Temporary variables are initialized to zero when a method is invoked. None have been declared for this method. Because temporary variables and arguments are so alike from a method's point of view (e.g., they are initialized and exist only while the method is executing) they are often grouped together and called *local variables*. Variables may have the same names as classes or methods—this may be useful in some cases where both are used to refer to the same thing.

The *pseudo-objects* self and super automatically refer to the receiver itself and the receiver of the superclass, respectively. The use of super enables a method to access the instance variables of the superclass—in this case, the *x*-coordinate Integer.

The object returned by the method is the result of the expression following the variable declaration. In this case, the dot operator and self are used to return the value of the Point receiver as the result. Although it is possible for methods to return no value at all by returning an Object, which has no instance variables, it is generally more useful for the method to at least return the receiver itself so that it can be used in an expression.

Class Hierarchy

In the following sections, some of the more sublime aspects of class hierarchies particular to object-oriented languages and pertinent to Grunt will be highlighted.

Overriding methods

It may be desirable to define a new implementation of the method + in Point in which the argument is added to both rather than just the *x*-coordinate. We can simply define a new method + within Point:

```
METHOD Point + Integer addend
||
super := super + addend.
y := y + addend.
self
ENDMETHOD
```

An even better version (like the one given in the appendix) would allow each coordinate of the point to be added to a different integer.

Method names must be unique in the class in which they are defined. Although it is allowable for a method to have the same name as a class, this practice is not encouraged because it may confuse the meaning (for the human, not the compiler). The Integer addend specifies the class and name of an argument of the message. A message may have any number of arguments:

```
"   Point set:
    Assigns to the Point
"
METHOD Point set: Integer newx, Integer newy ||
  super := newx.
  y := newy.
  self
  ENDMETHOD
```

The method is invoked as is suggested by its declaration, e.g. p1 set: 2, 3.

Now, instead of being passed on to Point's superclass Integer, the message + is understood and executed by Point itself. The superclass's method is said to have been *overridden* by a more effective method in the subclass.

Grunt allows messages to be overloaded in the sense that different classes may implement the same message in different ways. Grunt only allows this overloading of methods between classes—not within a class. The latter form would allow one class to respond to the same message in different ways, depending on the classes and number of arguments.† Smalltalk allows both types.

Data abstraction

In the previous examples, use has been made of the super pseudo-object to refer to the instance variables of the superclass. There is no way of accessing them other than through the sending of messages, as instance variables are private to a class and cannot be referenced directly, even from within subclasses. This may at first seem an unnecessary restriction, but in fact serves a useful purpose, as will be shown in the following example.

Consider a programmer who has implemented a class for complex number arithmetic called Complex, being a subclass of Integer. The programmer decides to represent its instances as points (x, y) in the complex plane, where either of the integer coordinates is inherited from Integer and the other is added as an instance variable. Because the instance variables in Complex cannot be accessed directly from other classes in the program, the programmer includes two methods real and imaginary which return the appropriate coordinate of a receiver. Eventually, other programmers start using Complex to implement their own, more capable classes.

Much later, the original programmer realizes that the implementation could have been much more efficient if he had used the polar (r, ϕ) representation instead. Instead of requiring all the other programmers to rewrite their classes (which would have been necessary if they had used instance variables in Complex), or maintaining two separate versions of the class, the programmer writes the more efficient version of Complex using polar coordinates and simply reimplements the real and imaginary methods, which will now require some trigonometric calculation.

Because of the *data abstraction* in Grunt, other programmers are forcibly shielded from the particular implementational details of other work. This concurs with the principle in object-oriented languages that new classes should build upon and add new functionality to existing ones.

Extensions

The following sections describe some language constructs which were part of the original Grunt definition but whose incorporation into the final language was precluded by budget and time constraints.

† This is like Ada, which differentiates procedures based on their formal parameter lists (called the *profile* of the procedure)

Blocks

In Smalltalk, *blocks* are a class of objects representing expressions whose evaluation is deferred until a later point in time. This feature of blocks makes them useful in the implementation of conditionals, as will be shown in the next section. *Block literals* are denoted as expressions enclosed in square brackets. Blocks respond to the message value by evaluating themselves, as in

```
[ x := x + 1] value
```

which has an identical effect to writing x := x + 1 directly. Naturally, literal blocks can be assigned to variable blocks:

```
incrementerBlock := [ x := x + 1].
" some other operations "
incrementerBlock value
```

The object incrementerBlock value is the one resulting from evaluation of the *block expression,* in this case, x := x + 1. In an implementational sense, blocks consist of subroutines and are represented by its entry point. Unfortunately, implementation of blocks poses two very serious problems in compiled languages.

First, in an interpreted language, determination of the class of an expression may be deferred until it is evaluated, and an error may then be generated if classes do not match. A compiler, however, must determine the classes of expressions *before* they are evaluated, namely at compile-time, so there is no “class-checking” at all when the program is run. The problem that arises is that although a block object is always an instance of the class Block (or whatever), the class of the block expression, that is, block value, depends, of course, entirely on the expression. How is the compiler to know then, if an expression like a := someBlock value is legal? One answer to the problem, and our preferred solution, is to restrict the class of the block expressions to some ‘neutral’ class, such as Object. Even though the block value itself does not return anything, meaningful results may be obtained through the side effects of the evaluation of the block, such as the changing of a local variable (as in incrementerBlock above). This brings us nicely to our second observation.

If blocks are to have any effect at all, they must have access to variables declared outside it—probably only those in the defining method, that is, the method in which the block is defined. The problem is, that a block which is returned as a result object by the method which declared it will produce strange results when it is evaluated, because the variables it references no longer exist. The situation does not arise in Smalltalk, because there the temporary variables and arguments exist in so-called *contexts* which are dynamically created by the run-time system and remain in existence even after the method has completed execution. A more practical solution for the Grunt compiler might have been to disallow the returning of block objects by methods.

Conditionals

Conditional selection of an activity can be implemented within the object-oriented framework without the introduction of additional symmetry-breaking mechanisms. Smalltalk provides one form of conditional selection through the sending of messages to Boolean objects:

```
condition ifTrueFalse: "true alternative", "false alternative"
```

The object condition responds to ifTrueFalse: by using the first argument if its value is true and the second if it is false. There also exist messages ifTrue: and ifFalse: which take only one argument which is used when the corresponding condition holds.

The arguments are not actually typical objects such as Integer—if conditional selection really were implemented in this manner, different versions of the messages would have to be written for each existing class. Instead, just one version of the method exists which takes block arguments.

Using blocks also solves a problem in the writing of recursive methods. Because *all* the arguments to a message are evaluated before it is sent, recursion never has an opportunity of stopping. This point is illustrated in the following implementation of the factorial operation in a blockless version of the language:

```
METHOD Integer ! ||                      " factorial—e.g., 3!  is 6 "
  (self = 0) ifTrueFalse:                " end of recursion? "
    1,                                   " 0! = 1 "
    self * (self – 1) !                  " n! = n*(n–1)! "
  ENDMETHOD
```

The second argument is always evaluated before ifTrueFalse: is sent, even when self = 0, so that this condition does not end the recursion.

With blocks, however, evaluation of the arguments involves only the interpretation (or compilation, as the case may be) of the block's code and passing it as the value of the argument. The actual evaluation of the block expression is deferred:

```
METHOD Integer ! ||
  ((self = 0) ifTrueFalse:
    [1],                                 " [1] value = 1 "
    [self * (self – 1) !]                " [self * (self – 1) !] value = n*(n–1)
! "
    ) value                                          " return the value of the
block "
  ENDMETHOD
```

This also increases efficiency with conditional statements in general, because expressions corresponding to false alternatives are not needlessly evaluated. Note that for this scheme to work, an exception would have to be made for Boolean to the rule that methods cannot return block objects. This poses no danger, because as it is a system class, we can trust it never to declare any blocks of its own.

Differences with Pascal

Observations on some of the differences between object-oriented and procedure-oriented languages may be helpful in determining the advantages of one type of language over the other, and in the understanding of the decisions made in the implementation of the language. To facilitate the discussion, Grunt and Pascal have been elected as representatives of their respective language types.

Communication

Pascal has both functions and procedures (although the latter may be seen as a special case of function), and arguments in Pascal can be both input or input/output. In Grunt, arguments are always input-only, the receiver is always input/output and a method always creates a result (although possibly one without contents, e.g. an Object object). The complete eradication of misunderstandings concerning the direction of communication or due to unexpected side-effects, although in some ways limiting, could be seen as an advantage of object-oriented languages.

Pascal allows local variables of a procedure to be used by procedures that are defined inside it. In contrast, Grunt limits the scope of temporary variables to just the method in which they are defined and the scope of instance variables to their defining class, increasing locality (thereby enhancing the ability to build large systems and easing debugging) without limiting functionality: methods can access other methods' instance variables by sending messages, so that classes always remain in complete control. It is like having data abstraction built in at every level, not just between modules.

2 Implementation

The implementation of the Grunt programming language on a VAX 8650 mainframe will be discussed. Also included are other important aspects including the various types of declarations in object-oriented languages and the generation of object code for the virtual stack machine for which the compiler is written. The significantly different scope of objects, as compared to procedure-oriented equivalents, and the different manner of expression evaluation led to the design and implementation of an almost completely new set of actions. This chapter of the report is divided into a part discussing the declaration of objects, methods, etc., and a second part which concentrates on object-code generation.

NIL procedure arguments are accepted and NIL results are returned by appropriate routines, usually to indicate that the argument or result is invalid and should not be used.

Definitions

The Grunt language includes several types of definitions—although some of them show resemblance to definitions in procedural languages like Pascal, they generally are somewhat different. The scope of local variables (temporary variables and arguments) in Grunt, for instance, is limited to the method in which they are defined—in contrast with Pascal where local variables are also accessible by other procedures defined in the procedure.

Instance variables have a scope limited to the class in which they are defined. This means that instance variables cannot be directly accessed from methods defined in subclasses (cf. the *Data abstraction* section in the previous chapter). Instance variables do not really have counterparts in Pascal.

Class and method definitions, however, have a wider scope than procedures in Pascal. The scope of a class or method is ideally the entire source code but is limited by the fact that declaration must precede use.

Finally, Grunt supports two other types of definitions which have been mentioned. Arguments of a message have the same scope rules as the temporary variables of the method, so they are treated the same by the Grunt compiler. It also must support the pseudo-objects `self` and `super`. This has important consequences for the data structures used to manage the definition data, as will be seen later.

Data structures

The identifier definition data structures employed by the parser's standard actions are not adequate for an object-oriented language—therefore, the data structures have been extended to support Grunt's more complex definition and scope structure. After careful examination of the structure of the definitions in Grunt, the following conclusions could be reached: first, the scope rules of instance variables, temporary variables and arguments have a sufficient resemblance to variables in a procedural language to permit the use of extended standard actions. However, the class and method definitions require a completely new definition mechanism and structure.

The definition data structure consists mainly of an enhanced `defdesrec`, the predefined definition descriptor record used by the standard actions. Many of the enhancements can be related to the hierarchical structure of the different types of definitions and to the strong-typed nature of Grunt. The `defdesrec` data type has been extended with a definition type-dependent variant record. The variant record is used, for instance, to link methods to their defining class, classes to their superclass and to store the result class of a method. It is not necessary here to present an exhaustive description of all the design decisions resulting in the definition data structure depicted below. However, many will become clear upon closer examination of the well-commented data structures, as well as the above and following descriptions of scope, object classes and overloading.

Creation

Definition data structures are created by calling one of the action procedures `NewClass`, `NewMethod`, `NewInstance`, `NewVariable` or `NewArgument`, each time a definition is encountered in the compiler input file. These procedures create and initialize new definition records, which are then linked into the definition data structure. Because it is a one-pass compiler, declarations precede use.

For reasons already given in the previous chapter, the current version of the compiler does not support recursive definition of methods—i.e., a method cannot be described in terms of itself; run-time stack overflow errors would occur if they were allowed. Another recursive, but meaningless, form which is not allowed is instance recursion. Both errors are detected at compile-time by checking whether an encountered message references the method which is currently being compiled, or whether the class of an instance variable is the defining class. This can be done quite easily because Grunt does not permit forward declaration of methods or classes. Would these have been allowed recursive constructions could be made which would have been much more difficult to detect.

Local and global definitions

Definitions in the Grunt language can be divided into *local* and *global definitions.* Local definitions can simply be added to the frontmost level of the definition data structure while global definitions must be added to a deeper level structure than those containing local definitions. For instance, method declarations are global and must remain in existence, while instance variable declarations are local and can be discarded outside the class. A special action `AddGlobalDefinition` was written which adds definitions to the deepest level in the definition data structure. The standard action `adddefinition` adds definitions to the frontmost level.

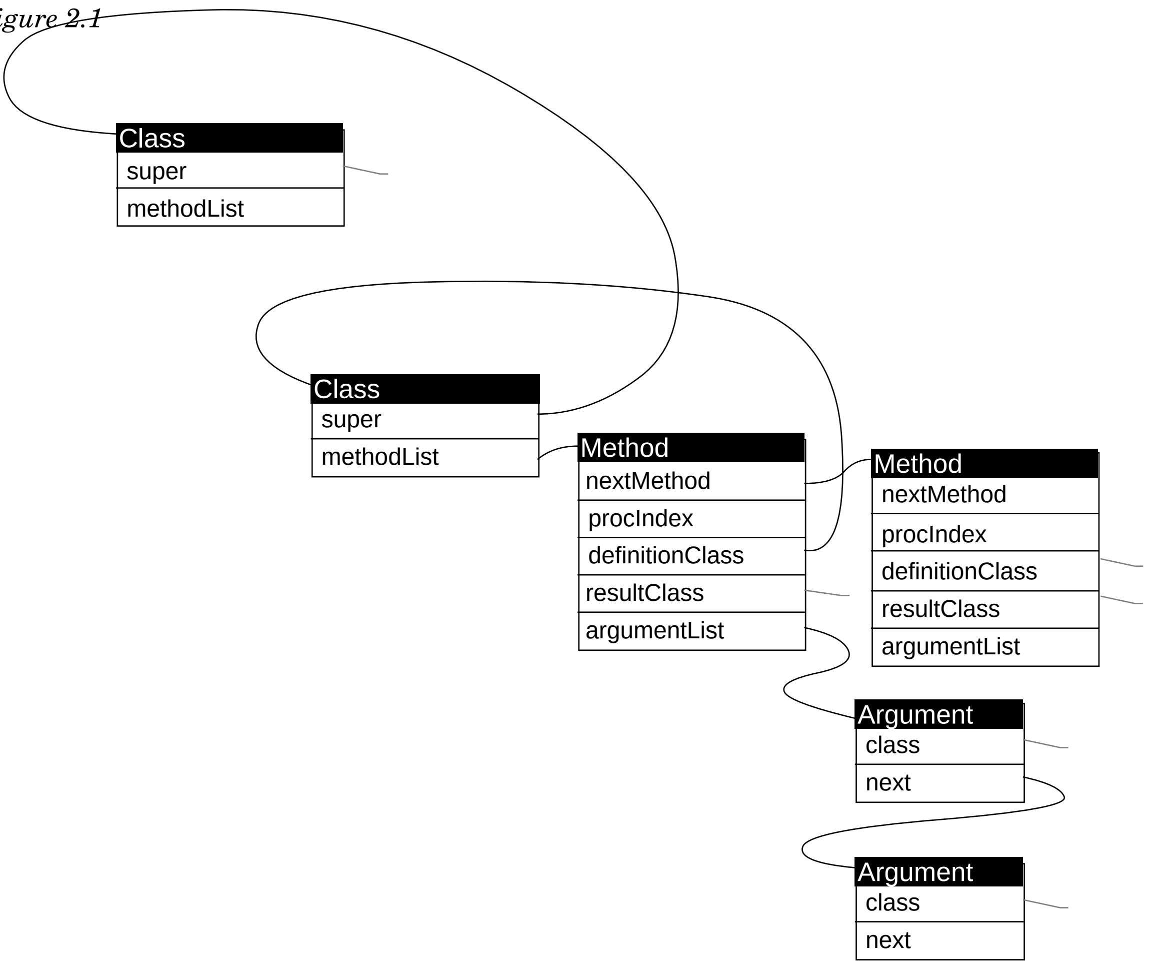
Grunt uses two levels of local definitions. The first level is created every time a class definition is entered. It contains the definitions of the instance variables of that class. This level remains in existence during the compilation of the methods of that class. After the last method of a class is compiled and before optional subclasses are compiled this level is removed from the definition structure. The second local level is created every time a method definition is entered and destroyed upon leaving the method. Argument and temporary variable definitions are contained in this level. Arguments are actually also given a global declaration, consisting of a small record which is added onto a method's *argument list.* The argument list is traversed when the message is sent to type-check the classes and number of arguments.

The global level is not necessary in procedure-oriented languages because the order in which definitions are allowed in Grunt differs from that in procedure-oriented languages. In Pascal, for instance, the nesting of definitions strictly dictates their scope—in Grunt this is not always the case. Class and method definitions are always global and remain in existence even when their definitions have been left.

Definitions have *defining classes* and *defining methods* which are the method or class containing the definition. The Grunt compiler uses various attributes to pass the defining classes and methods through. At the top of the definition hierarchy is the Object class which has no defining class at all.

More details about how definitions are dealt with by Grunt can be found in the listings of the files `gruntpars.inp`, `defown.pas`, and `initown.pas` [L2, L4, L5]. Global definitions in Grunt are depicted in the following figure:

Figure 2.1



Not shown in this figure is the fact that synonymous `defdesrec` definitions are also linked.

System classes

Just prior to the generation of the code for the user methods the compiler declares the are system classes and primitive methods using `NewClass` and `NewPrimitiveArgument`. The code for the primitive methods is also generated at this time. Following the initialization, system classes and primitive methods are handled in precisely the same manner and are indistinguishable from other definitions.

Code generation

The ‘functional’ part of the Grunt language consists solely of expressions. Expressions are evaluated by sending messages to objects. Grunt code consists of two parts. As stated, code implementing the primitive methods is generated at the start of the compilation process. This is the only place where instructions such as ‘add’, ‘compare’ or ‘and’ occur—they do not occur in the code generated by user methods. This is an interesting property which would allow programs to take advantage of new machine instructions without having to be recompiled. Imagine for instance a processor for which a software emulation for floating-point arithmetic is implemented in primitive methods. When a new version of the processor is used which includes floating-point hardware, simply changing the code of some primitive methods is sufficient to let all programs take advantage of the new hardware.

The other code concerns that generated by the compiler from the source file, which is the type described in this chapter.

Message sending

Grunt's equivalent of calling procedures in Pascal is the sending of messages by methods. In this section, the protocol used to invoke a method will be discussed. Broadly, this protocol must concern itself with the following requirements. The value of the receiver of the message may be altered by the method. It is possible for a method to accept arguments, which are treated as initialized temporary variables and are input-only (in other words, if a method changes the value of its arguments the value of the sending method's argument objects do not change). After the method has completed it returns a result object.
Besides the two types of objects that are passed by the calling method, a method has six other objects which can be accessed. A method can have temporary variables, and access its defining class's instance variables. The super pseudo-object can be used to send messages to the superclass of the receiver's class, and self can be used in an expression to reference the receiver object itself. Two types of unnamed objects, literal and temporary, also occur. Literals are objects which are defined when they are used and cannot be referenced elsewhere. Temporary objects are created during the evaluation of subexpressions and do not occur explicitly in the program text.

A design has been used that employs a single stack for all these types of objects. Using a second stack to store literals and temporaries has some advantages but is more difficult to implement on an actual processor. As it is, a *stack frame* is created upon entry of a method and deleted when it is exited. The choice of the format of the stack frame has been based on several aspects of the Grunt language. Because the scope of temporary variables is strictly the defining method, static links become unnecessary in Grunt.

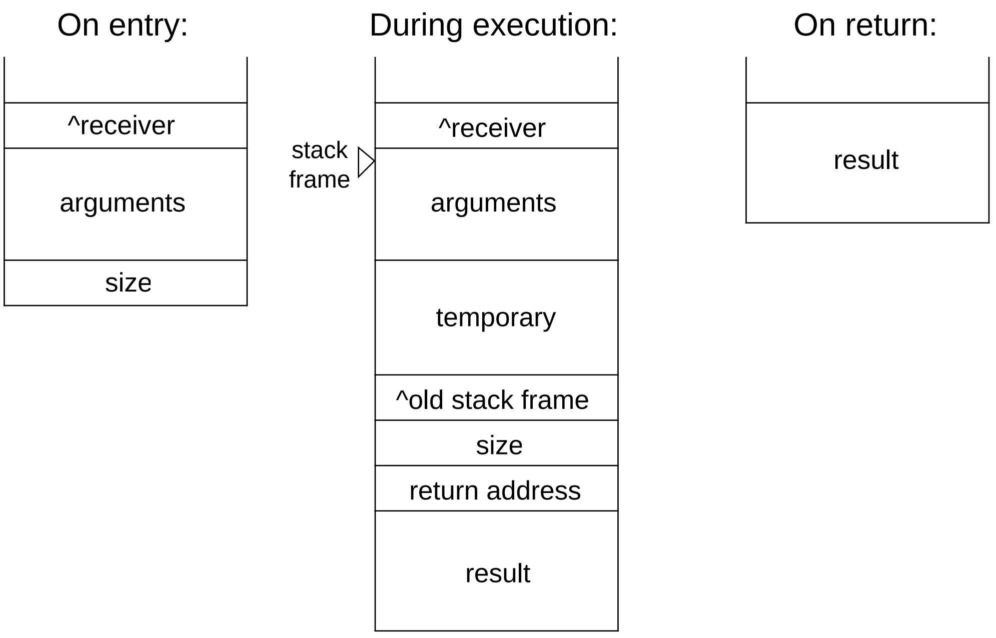
For this implementation of Grunt we choose to use a hybrid method invocation convention (see figure 2.2). Since receivers can always be modified by methods, a pointer to the receiver object is passed to the method (analogous to call-by-reference). This makes it unnecessary to copy a possibly modified receiver back to the original object after the method has completed. Arguments, being input-only, are pushed onto the stack and are removed by the called method together with its temporary variables at its completion. The result object of a method is moved to the top of the stack. This approach has the advantage that using expressions as arguments poses no problems, since expressions also leave just a result object on the stack.

The fact that we pass receivers by means of a reference implies that the receiver object must already exist somewhere, but unfortunately, this is not always the case. Literal receivers must be pushed on top of the stack when they are encountered and then removed after the method has executed. The latter also holds for temporary objects. By passing to the method the size of objects to be removed this can also be performed by the method at the same time as the removal of its own variables. Having both reference and value parameters also has an instructional advantage.

Expressions in Grunt are grammatically defined as sequences of subexpressions, called *item expressions,* separated by periods. Because the result of an expression is the result of the last item expression, all but the last of the result objects of item expressions must be removed from the stack.

Some of the complexity of stack management in Grunt can be traced back to the fact that object-oriented languages generally employ dynamic memory management to store objects and contexts and therefore do not suffer from LIFO-restrictions inherent to stack-based mechanisms.

Figure 2.2



Instructions

Several new instructions had to be introduced in order to implement the proposed object code structure. For simplicity, each of these instructions takes one instruction selector and two arguments. Only the arguments that are actually used by the instruction are shown—the others are ignored.

Object code instructions in Grunt programs can be grouped into three sets: the first set contains two instructions used to create and delete stack frames. Instructions used to implement the sending of messages, which forms the foundation of Grunt, are contained in the second set. The last set contains some special instructions that are used in the implementation of the primitive methods.

The main Pascal source code modification to the interpreter is given in the partial listing of `owninter.pas` [L6].

Stack frame maintenance

`iuser 1 argSize tmpSize` `create frame`

The *create frame* instruction creates a new local stack frame upon entry into a method. It allocates the temporary variable space whose size is specified by `tmpSize` and initializes them to zero. `argSize` is the size of the arguments on the stack. It also sets the stack frame pointer to point to the first argument after saving its previous contents, as well as the return address.

`iuser 2 varSize resultSize` `delete frame`

The function of the user instruction *delete frame* is to undo the actions of the previous instruction, that is, to delete the local stack frame, and to return from the method. It copies the result object to the appropriate location on the stack, over any literal or temporary object that was used as receiver for the message, and cleans up the stack by disposing of the arguments and temporary variables. Execution resumes at the saved return address.

Message sending

The set that implements the message sending must be able to push pointers to all sorts of objects on the stack, as well as pushing the objects themselves when they are used as arguments. In particular, instructions are needed to access receiver objects, instance variables, temporary variables, arguments, temporary objects and literal objects. A more detailed consideration shows that all these objects can be accessed using just three ‘addressing modes’, resulting in what appears to be an almost-orthogonal instruction set. The instruction `iuser 4` is not used.

`iuser 3 size` `pop receiver`

Pops the object from the stack and stores it in the receiver object, a pointer to which is located just before the stack frame pointer. `size` is the size of the object.

`iuser 5 offset size` `push variable`

Pushes the local variable with the specified size and offset from the stack frame pointer onto the stack. Note that the displacement field (`dpl`) of argument and temporary variable declarations already contains the correct offset. The size can be retrieved from the displacement field of the `variableClass` of the variable.

`iuser 6 offset size` `push instance`

Pushes the instance variable with the specified size and offset within the receiver object onto the stack. Note that the displacement field of instance variable declarations already contains the correct offset. The size can be retrieved from the displacement field of the `instanceClass` of the variable. The entire receiver object can be pushed using an offset of zero and using the size of the receiver class.

`iuser 7 offset` `push address variable`

This instruction pushes a pointer to the local variable with the specified offset from the stack frame pointer onto the stack. This instruction is used when a local variable is the receiver of a message. See also the note for the `iuser 5` instruction.

`iuser 8 offset` `push address instance`

This instruction pushes a pointer to the instance variable with the specified offset within the receiver object onto the stack. This instruction is used when an instance variable is the receiver of a message. See also the note for the `iuser 6` instruction.

`iuser 9 size` `push address temporary`

Pushes a pointer to the temporary object currently on top of the stack. This instruction is used when a temporary result is the receiver of a message.

`iuser 10 size` `discard`

Destroys the object on top of the stack. This instruction is used when an evaluated item expression is followed by a second to delete the result of the former.

Special instructions

The instructions in this final group are used in the implementation of the primitive methods, such as addition (`iadd`), reading (`iread`) and writing (`iprint`). These are sufficiently well documented in [4, pp. 83-94].

A
Examples

The following example illustrates the declaration of class Point and Rectangle and some methods. The compilation of this file is given in `grunt.tgc` [L8]. A sample run of the program is given in [L9, L10]. A stack trace of the x: method is listed in [L11]. Finally, the compiler listing is shown of a typical Grunt program `faulty.inp`, as a beginning object-oriented programmer might write it.

```
"
  GRUNT.INP

  Example of Grunt source code

  Version 1.2       06/03/90

  (c) Copyright 1990 by Ben Hekster
"

"
  Point class definition

  Inherited         : Integer x
  Instanced         : Integer y
"

CLASS Point PARENT Integer
| Integer y |

  "   Point x / x:
    Returns/sets the horizontal coordinate of the Point
  "
  METHOD Integer x || super ENDMETHOD
  METHOD Point x: Integer new || super := new. self ENDMETHOD


  "   Point y / y:
    Returns/sets the vertical coordinate of the Point
  "
  " These examples demonstrate that it is OK to define messages with
    the same names as variables, as Grunt can always tell which of
    the two we mean! "
  METHOD Integer y || y ENDMETHOD
  METHOD Point y: Integer new || y := new. self ENDMETHOD


  "   Point :=
    Assigns to the Point
  "
  METHOD Point := Point new ||
    super := new x. y := new y. self
    ENDMETHOD


  "   Point +
  Adds the specified point
"
" We could have called this one 'translate', but we like to retain
  consistent message nomenclature across classes.
  Note that we implement this method by sending messages
  to self, which must be defined before this point "
METHOD Point + Point delta ||
  " The inner set of parentheses is used to create a new object
    The outer set is needed to prevent Grunt from thinking
    y: is a message to x, i.e. delta (x y: y) "
  ((self) x: super + delta x) y: y + delta y
  ENDMETHOD


"   Point read
  Reads the Point as an 'x y' coordinate pair
"
METHOD Point read ||
  super read. y read. self
  ENDMETHOD


"   Point write
  Writes the Point as 'x,y'
"
METHOD Point write ||
  super write. $,$ write. y write. self
  ENDMETHOD


"
  Rectangle class definition

  Inherited:          Point topLeft (left, top)
  Instanced:          Point bottomRight (right, bottom)
"
  CLASS Rectangle
  | Point bottomRight |

    " Of course, we could also have defined methods to return
      all four of the corners, or alternatively, the top, left,
      bottom, right coordinates "

    "   Point topLeft / topLeft:
      Returns/sets the (left, top) corner of the Rectangle
    "
    METHOD Point topLeft || super ENDMETHOD
    METHOD Rectangle topLeft: Point new ||
      super := new. self
      ENDMETHOD


    "   Point bottomRight / bottomRight:
      Returns/sets the (right, bottom) corner of the Rectangle
    "
    METHOD Point bottomRight || bottomRight ENDMETHOD
    METHOD Rectangle bottomRight: Point new ||
      bottomRight := new. self
      ENDMETHOD


    "   Rectangle :=
      Assigns the Rectangle
```

```
          "
          METHOD Rectangle := Rectangle new ||
            super := new topLeft.
            bottomRight := new bottomRight.
            self
            ENDMETHOD


          "   Rectangle read
            Reads the left, top, right, bottom coordinates into the Rectangle
          "
          METHOD Rectangle read ||
            super read. bottomRight read. self
            ENDMETHOD


          "   Rectangle write
            Writes 'left, top, right, bottom'
          "
          METHOD Rectangle write ||
            super write. $,$ write. bottomRight write. self
            ENDMETHOD


          ENDCLASS " Rectangle "

        ENDCLASS " Point "


      "
        Program class definition

        Inherited          : Object
        Instanced          : none
      "
      " Now we must write a method for the Program class, so the Grunt
        compiler will know what to run. Note that without specifying one,
        Program's superclass automatically becomes Object "
      " Returning the super Object is a tricky way of avoiding any
        unneccessary data transfer when we know we don't want to return
        anything anyway
      "
      CLASS Program
      ||

      METHOD Object runIt
        | Point p. Point delta. Rectangle r |
        (p x: 2. p y: 3) write.
        (p + delta read) write.
        $$n$ write.
        r topLeft: p. (r bottomRight: (p) read) write.
        p write.
        super
        ENDMETHOD

      ENDCLASS
```

The translation of this example is given in [L7].

In this example, many of the methods of the superclasses needed to be overridden, which does not make a very good case for the merits of method inheritance. Its value may become clear, however, in the following implementation of ColoredPoint:

```
CLASS ColoredPoint PARENT Point
| Integer color |

  METHOD Integer setcolor Integer newColor ||
    color := newColor
    ENDMETHOD

  METHOD Integer getcolor ||
    color
    ENDMETHOD

  ENDCLASS
```

All of the methods defined in Point will work as expected—that is, assigning to, writing or translating a ColoredPoint still produces the same result as to a Point. This permits the completely painless substitution of a ColoredPoint in place of Point. Wherever needed, the getcolor and setcolor messages can be used.

Finally, the following sample file was offered to the Grunt compiler:

```
CLASS Program
| Integer y. Point x. Program prog |

METHOD Integer := Integer x, Integer x
  true not
  ENDMETHOD

METHOD Integer aRealMethod Boolean b, Integer i || i ENDMETHOD

METHOD Integer recursive
  | Program p |
  p recursive.
  p aRealMethod 3, true, $a$.

ENDCLASS
```

The resulting output is shown in [L12].

B
Grammar

Following are the formal syntax and representation rules of the Grunt language†. They have been derived from [L1, L2].

```
grunt           : classlist.
classlist       : class CLOS.
class           : classsym, classname, (parentsym, classname) OPTION,
                  instancelist,
                  methodlist,
                  classlist,
                  endclasssym.
classname       : identifier.
instancelist    : declsym, (instance CHAIN separsym) OPTION, declsym.
instance        : classname, variablename.
variablename    : identifier.
methodlist      : method CLOS.
method          : methodsym, classname, methodname, argumentlist,
                  declarationlist,
                  expression,
                  endmethodsym.
methodname      : identifier.
argumentlist    : argument LIST OPTION.
argument        : classname, argumentname.
argumentname    : identifier.
declarationlist : declsym, (declaration CHAIN separsym) OPTION, declsym.
declaration     : classname, variablename.
expression      : itemexp CHAIN separsym.
itemexp         : receiver, message OPTION.
message         : methodname, parameterlist.
parameterlist   : startsym, itemexp LIST OPTION, stopsym.
receiver        : opensym, expression, closesym;
                  variablename;
                  selfsym; supersym;
                  literal.
literal         : integersym;
                  charsym;
                  truesym; falsesym.
comment         : commentsym, (character - commentsym) SEQ, commentsym.
identifier      : (letter; symbol), (letter; digit; symbol) CLOS.
integersym      : digit SEQ.
charsym         : charlitsym, charlitsym OPTION, character, charlitsym.
letter          : a, .., z, A, .., Z.
digit           : 0, .., 9.
symbol          : !, #, %, &, *, +, -, /, :, ;, <, =, >, ?, @, \,
                  ^, _, `, {, |, }, ~.
classsym        : CLASS.
parentsym       : PARENT.
endclasssym     : ENDCLASS.
methodsym       : METHOD.
endmethodsym    : ENDMETHOD.
selfsym         : self.
supersym        : super.
truesym         : true.
falsesym        : false.
declsym         : |.
```

† Actually, they apply to the unmodified version of the language, not the final

```
separsym        : ..
opensym         : (.
closesym        : ).
startsym        : [.
stopsym         : ].
charlitsym      : $.
commentsym      : ".
```

The nonterminal `comment` has been included in this definition, even though it does not appear in the right-hand side of any other nonterminal. Its legal place is between any two nonterminals.

This definition of the Grunt grammar includes a definition of the `parameterlist` nonterminal enclosed within square brackets (`startsym` and `stopsym`), which were needed to satisfy the LL(1)-property of the syntax. Consider the statement

> receiver1 message1 receiver2 message2 argumentA, argumentB

It is not clear to which message the arguments belong—they could be interpreted as either of:

> receiver1 message1 [receiver2 message2 argumentA, argumentB]
> receiver1 message1 [receiver2 message2 argumentA], argumentB

This is called the *argument ambiguity*.

Because we wanted to emulate the Smalltalk syntax as much as possible we needed to transcend the boundaries of LL(1)-grammars. During development we made use of the original definition described above which required parameter lists to be enclosed within brackets. After we were sure no additional changes to the parser were necessary, the parser was recompiled with the state procedure representing the `parameterlist` nonterminal substituted by the following modified version:

```
{    p_param092
     Modified 'parameterlist' nonterminal parser
     Does not require the parameter list to be enclosed within
     square brackets--instead, reads as many arguments (separated
     by commas) as possible
}
PROCEDURE p_param092; {(keys: tsymbolset; VAR v_par09200: t_param092)}
CONST
     receivers               = [opensym, charsym, integersym, identifier,
                             falsesym, selfsym, supersym, truesym];
VAR  theargumentlist         : argumentlistptr;
     v_ite08901              : t_iteme089;
BEGIN
firstargument(theargumentlist, v_par09200.plmethod);
{ next symbol may be a receiver }
delete(keys + receivers);
IF sym IN receivers THEN REPEAT
     { there IS a receiver }
     IF sym = commasym THEN nextsym;
     assdefinition(v_ite08901.iedefinitionclass,
                   v_par09200.pldefinitionclass);
     { receiver may be followed by a ',' }
     p_iteme089(keys + [commasym], v_ite08901);
     checkargument(theargumentlist, v_ite08901.ieclass)
     UNTIL sym <> commasym;
lastargument(theargumentlist);
delete(keys);
END;
```

Testing then showed the above procedure to give the desired results. The only change to the syntax given above is

```
parameterlist   : itemexp LIST OPTION.
```

Of course, using this 'rule' makes the grammar non-LL(1). The terminals `startsym` and `stopsym` might have been moved to `symbol` to give Grunt programmers some extra flexibility—note, however, that they are also used in the implementation of block literals.

The compiler evaluation rule is, then, that arguments always belong to the last (rightmost) message. Therefore in the example the first alternative is the correct one. A more sophisticated compiler might deduce from the message profile to which message an argument belonged.

Note that similar situations exist in procedural languages. For example, in the following C statement:

```
if (condition1) then
     if (condition2) then
          statement1;
else
     statement2;
```

it is not clear whether the `else` clause applies to the first or second `if-then`. This so-called *else ambiguity* is resolved by always connecting an `else` to the last `else`-less `if` [5, pp. 56, 223].

C Translation

Following are the translation rules for Grunt. As in [6, p. 160], curly brackets ‘{}’ refer to items which may be repeated zero or more times. Also, in the translation of the nonterminal `grunt`, `InitGrunt` refers to the translation of the primitive methods in that action procedure. The code which executes the call to the first method in `Program` which is also generated in this routine is listed explicitly. For conceptual clarity, the `itemexp` nonterminal has been split in its occurrence with and without a trailing `message`.

```
<grunt>          : InitGrunt,
                   ildint(0), ildint(0), icall(startLabel), istop,
                   <classlist>.
<classlist>      : {<class>}.
<class>          : <methodlist>.
<methodlist>     : {<method>}.
<method>         : iuser(1, argSize, varSize),
                   <expression>,
                   iuser(2, argSize+varSize, resultSize).
<expression>     : <itemexp>,
                   {iuser(10, expClassSize, 0), <itemexp>}.
<itemexp>        : <receiver>,
                 ( ;
                   iuser(5, dpl, size);
                   iuser(6, 0, size);
                   iuser(6, dpl, size)
                 ).
<itemexp, message>
                 : <receiver>,
                 ( iuser(7, dpl, 0);
                   iuser(8, dpl, 0);
                   iuser(9, size, 0)
                 ),
                   ildint(size), icall(methodIndex).
<receiver>       : <expression>; <literal>.
<literal>        : ildint(intValue);
                   ildchar(charValue);
                   ildint(1);
                   ildint(0).
```

It will be clear that much of the regular simplicity of a compiled Grunt program is derived from the fact that all the primitive (built-in) operations normally associated with programming languages are treated in precisely the same way as user-defined operations, whereas typical procedure-based languages require special translation rules for each of its operators.

In many instances it might appear more appropriate to move some of the translation to other nonterminals. For instance, the instructions for pushing the value of temporary variables, receivers, or instance variables in `<itemexp>` might seem better placed in `receiver`. This, and other cases like it, could not be implemented in this manner, because at the time `receiver` is being parsed it is not known whether it will be followed by a `message` or not. The translation must therefore be deferred.

D
Bibliography

The [L]-references apply to the listings accompanying the report.

[1] Goldberg, Adele, David Robson, *Smalltalk-80, The Language and its Implementation,* Addison-Wesley

[2] *Smalltalk/V Mac Object-Oriented Programming System (OOPS),* Digitalk Inc., Los Angeles, September 1988

[3] Storrie-Lombardi, Michael C., *Smalltalk/V Mac: A New Standard in Object-Oriented Programming,* MacTech Quarterly, Volume 1 Number 2, Summer 1989

[4] Schaap-Kruseman, J.P., *Handleiding scanner/parser generator en praktikum vertalerbouw,* juni 1989

[5] Kernighan, Brian W., Dennis M. Ritchie, *The C Programming Language,* Second Edition, Prentice-Hall

[6] Alblas, H., et al., *Vertalerbouw,* Academic Service

Index