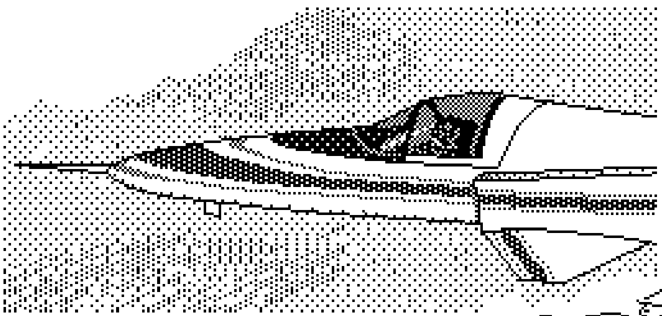
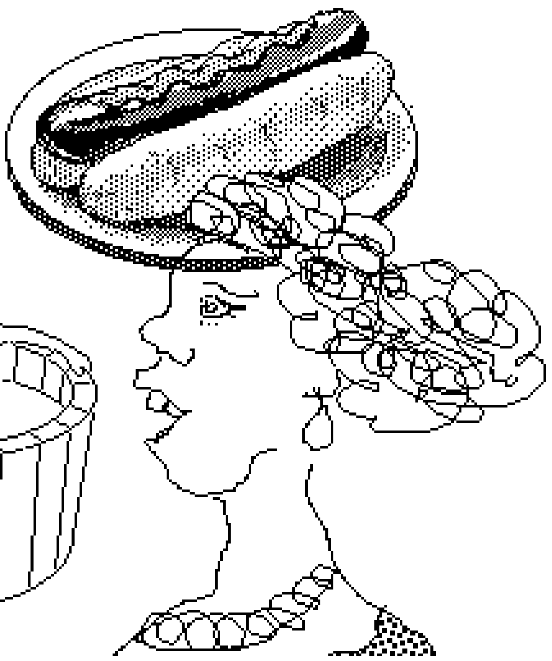
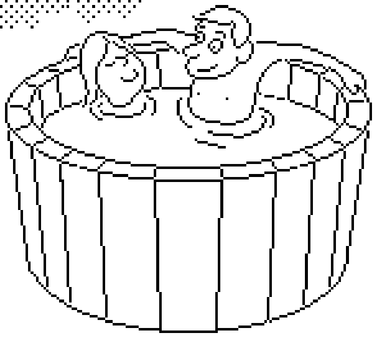
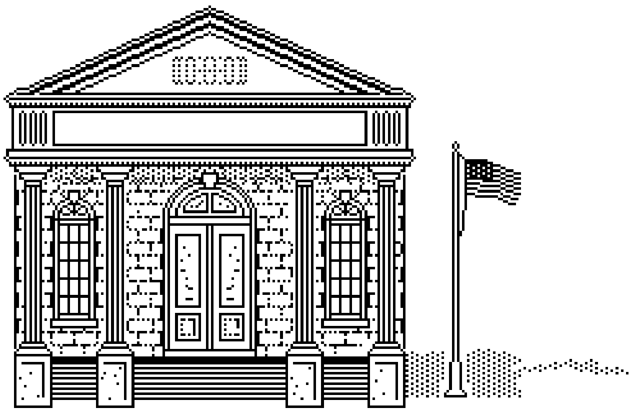
# *The Programming Language* GRUNT-91



Ben Hekster and
Peter Middelhoek

# Preface

Before you is the report on the *Grunt-91* programming language which was written as the assignment for the course *Special Topics in Compiler Design* (*Bijzondere Onderwerpen Vertalerbouw*). As its name suggests, Grunt-91 is an updated version of the *Grunt* language (henceforth referred to as Classic Grunt) which resulted from the original course in Compiler Design (*Vertalerbouw*).

Aside from its complete reimplementation under a different compiler-building environment, Grunt-91 embodies several (we feel) significant and some minor changes with respect to the original language. A second pass through the language and compiler design process has greatly improved our understanding of compilers and object-oriented languages.

▼ This document is a very thorough reworking of *The GRUNT Programming Language* [1], which described the original Grunt language and an implementation. Features new to Grunt-91 or significantly different from Classic Grunt are set off like this paragraph. Those already familiar with the original document or language may wish to concentrate on these points only.

## *The language*

Grunt was inspired by and derives its name from the visionary Smalltalk-80 language developed at the Xerox Palo Alto Research Center [1]. Many of its basic concepts were 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.

Upon reiterating the design process for Grunt-91, and with the benefit of hindsight, some incorrect assumptions we made due to our incomplete understanding of object-oriented languages have become apparent. Our earlier assertion that object-oriented languages are mostly interpreters was, considering the rising popularity of languages like C++ somewhat imprecise.

Since the language has now been completely reimplemented using the widely available LEX and YACC compiler-generation tools, we foresee being much more readily able to extend the language or port the compiler after completing the course. We regret not having had this opportunity in the first place.

We are happy to say that in spite of many changes the language is still type-safe, meaning that Grunt programs do not have type conflicts after they are compiled. We believe that determining the validity of a program should as much as possible be done at compile-time and not be deferred at the expense of the program's users every single time it is run.

In a sense, Grunt carries on a trend which has been set by languages such as C, of removing extraneous concepts from the language itself. For example, the C input/output facilities are not part of the language proper, but can be included in the form of a standard library. In Grunt even control constructs such as loops or conditional execution are not part of the language but can be implemented as system or library classes. A language must be flexible enough to allow such functionality to be implemented.

## *The report*

Contrary to the original document, the formal specification of the grammar has been moved from its separate appendix and merged into the informal description. This permits a more precise and comprehensive treatment of the language's features and peculiarities. Unfortunately this also often means that the reader must understand certain concepts before they are defined. The report thus takes on more the form of a reference manual, in the style of [9].

It no longer presumes to serve as any kind of introduction into object-oriented languages. The reader is assumed to be familiar with their concepts and terminology. A small introductory section on the object-oriented programming model is included mainly to clearly define and justify our own use of the terminology, which we have seen to vary somewhat even among well-established object-oriented languages.

In one way this report shows a maturation of our approach to language design—whereas before we took a myopic view of the language and its implementation being one and the same, we now view them as distinct entities. Meaning, that we have tried not to let deficiencies in one particular implementation compromise the definition of the language. Naturally differences are noted when they occur. A chapter still describes the implementation of the language but on a more platform-independent and instructive manner so that it might serve as a guide to future implementations.

Typographical conventions in this report are:

▼ Departures from Classic Grunt

▲ Suggestions for future extensions to Grunt-91

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.

## *Observations*

In developing the Grunt languages we have very much relied on knowledge developed by others. References are made to literature throughout this report. In particular we are indebted to [1] and [9], which are both excellent and have set the tone of this report. Although interesting, the 'dragon book' [10] unfortunately does not seem to have lent itself particularly to the issues of object-oriented languages and their compilers.

To us the ultimate test of a language and compiler is an implementation of the compiler in its own language. Although there is still some way to go, Grunt-91 takes a step towards this goal.

July 1991

*Ben Hekster*
*Tankelanden 5*
*7542 DR Enschede*
*053-764091*

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

# Contents

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

# 1

# Introduction

This chapter gives a brief demonstration of the terms used to describe Grunt-91's object-oriented concepts. Lexical conventions of Grunt-91 programs are defined, and there is a short section of compiler-specific information.

## Object Paradigm

In an object-oriented system, an **object** is often defined as being a collection of private data together with some set of operations which can be performed on it. In mature object-oriented languages, examples of objects are integer numbers, 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 to which the message is directed is called the **receiver** of the message. The object's implementation of the operation is called its **method**. 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 example, 3 is the receiver of the message + a. The integer object a is the **parameter** of the message. Methods can return the result of an operation by changing the receiver, or by returning a new *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. The text between the double quotes is a *comment* which 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, sending a message to an integer object asking it to compute its Boolean negation constitutes an an error:

```
3 not
```

An object whose value is explicitly stated is called a **literal**—for example, 3 is an integer literal. The $ symbol denotes a character literal, as in $a. There are also two Boolean literals true and false. Square brackets [] are used to introduce a so-called *block literal* (explained later on). The object class of a literal is hereby unambiguous even though it is not spelled out in the program.

Their counterparts are the **variable objects**, which are not specified by a value but by an identifier which is called its **name**. A variable must be declared before it can be used, which involves specifying its object class and an optional initial value.

Objects usually understand an assignment message := which can be used to assign a new value to an object, as in

```
a := 2
```

The terms “literal” and “variable” are actually somewhat of misnomers since, while they exist, literal and variable objects behave identically. For example, a literal object can also be sent a message to change its value:

```
3 := 2
```

Literals are declarations of objects which belong to a type called **temporary objects**. These describe short-lived unnamed objects whose creation and disposal is not under the user's control but is managed implicitly by the compiler. The other types of temporary object arise in the concatenation of expressions, and when expressions are enclosed in parentheses. A temporary object exists only within the expression in which it occurs.

The implicit-declarative nature of literals has two consequences. Because its object class is not explicitly specified the class of literals that can be written is extremely limited. Second, because a literal is unnamed it can only be referred to at the point it is written.

▲ A future extension of the language could include a capability for specifying literals of arbitrary object classes

The fact that literals do not require explicit declaration can occasionally be useful. For example, to input an object directly into an expression we could write:

```
0 (:read) :write
```

without needing to explicitly declare a variable object. A value is read into the integer object 0 which is then displayed on the output. The exact interpretation of an expression such as this is explained later on.

## Lexical Conventions

Except when preceded by a dollar sign, the double quote character " introduces a **comment** which terminates with the next double quote. Comments can therefore not be nested. **White space** consists of the space, tab and newline characters. Other than that they are used to separate tokens they are ignored.

**Identifiers** consist of an alphabetic letter (upper and lower character cases are distinct) or one of the symbols

```
! # % & * + - / : ; < = > ? @ \ ^ _ ` { } ~
```

followed by any number of alphabetic letters, symbols or digits.

The following identifiers are reserved as **keywords** and cannot be used otherwise:

```
CLASS        self
ENDCLASS     super
METHOD       true
PARENT       false
GENERIC
```

In addition, the keyword

```
PRAGMA
```

at the beginning of a line introduces an implementation-dependent action on the remainder of that line, which is otherwise ignored and not part of the program. Unrecognized pragmas are ignored.

There are four kinds of literals corresponding to four of the s*ystem classes*. An integer literal consists of an optional plus or minus symbol followed by a sequence of digits, representing a signed decimal value. A Boolean literal is denoted by either of the keywords `true` or `false`. The left and right square brackets [] enclose block literals.

A character literal is specified by a dollar sign $ followed either by a single character or another dollar sign and a single character. The latter form specifies one of the following **character escape sequences**:

| | |
|---|---|
| `$$n` | newline |
| `$$t` | tab |
| `$$$` | dollar |

Use of escape sequences other than those defined above is allowed and is the same as if the character had not been escaped, but results in a compiler warning. This provides some backward compatibility of programs written under future versions of the language which define additional sequences.

## Pragmas

Borrowing one of the ANSI C preprocessor features, pragmas are used for passing information to the compiler. Grunt-91 requires the following pragma to be recognized by all its implementations:

```
PRAGMA INCLUDE file
```

causes the named *file* to be parsed in place of that line in the program.

▼ Pragmas are new to Grunt-91

▲ Including files would be more useful if class declarations could be separated from their definitions.

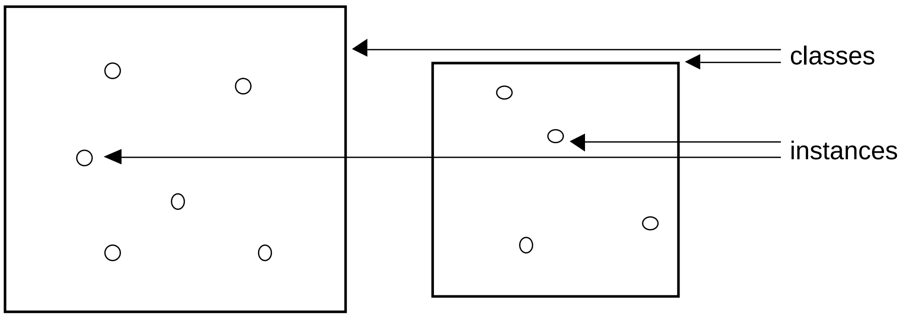
# 2

# Classes

Same kinds of objects are grouped into **classes** (see figure). Individual objects belonging to a class in this way are called **instances** of the class, which is known as the instance's **object class**. Instances have their own copies of the private object data, called the **instance variables** (or **object variables**).

A class serves a twofold purpose—first, it defines the format of and initializes instances of itself. Second, it contains the class variables, which include the implementation of their methods.

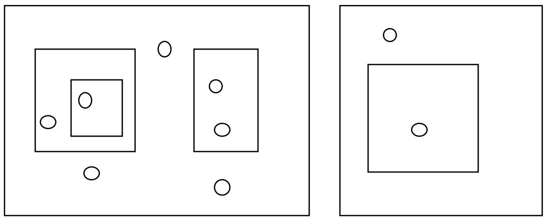
*Figure 2.1*



## *Superclasses*

New classes can be defined in terms of existing classes by specifying additional object variables and class methods. 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 2.2*

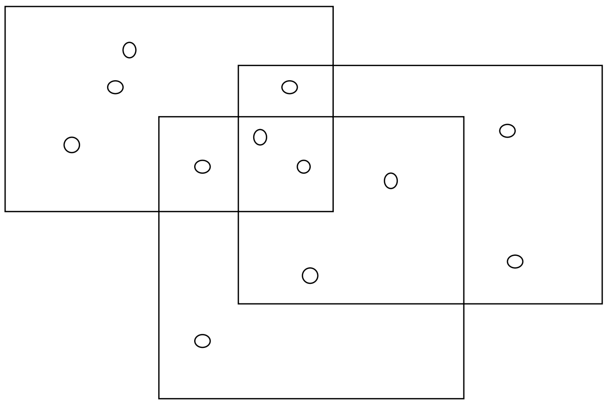


A class can have many immediate subclasses but has only one immediate superclass. There is exactly one class, Object, which has no superclasses at all. All other classes are either direct or indirect subclasses of this class.

The subclass **inherits** all of the object variables and methods of its superclass—it builds on and implements functionality absent in the superclass. In this way, messages can be sent to instances of a subclass which an instance of the superclass would not understand. Messages sent to an instance of a subclass which the subclass does not understand are passed on to the superclass. If a subclass reimplements a method of one of its superclasses, the method is said to **override** the one in the superclass. The overriding method may relay the message to the superclass while performing some additional actions, or it may reimplement the message response completely.

▲ Multiple inheritance should be implemented in another version of the language. In this scheme, objects can belong to more than one class at the same time. For instance, a new class with Boolean and Integer superclasses would inherit data and behavior from both these classes (see figure).

*Figure 2.3*



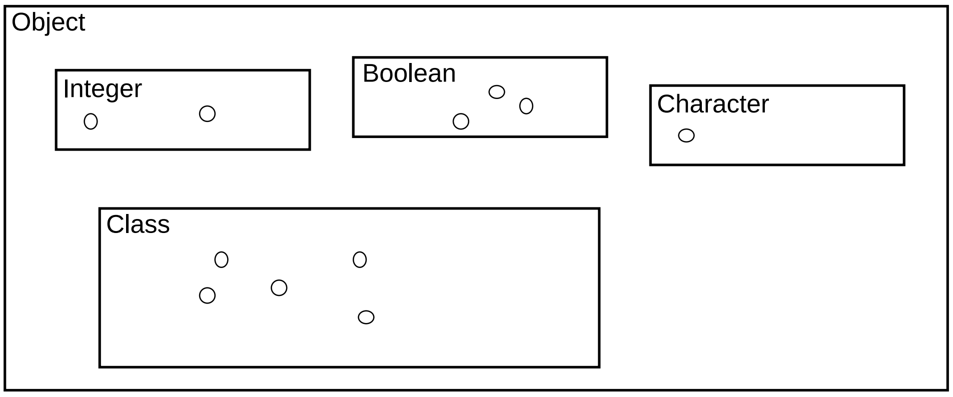
## *Metaclass*

A class is itself an object. Its object variables are the class variables of its instances and include the class methods.

Classes are instances of a single **metaclass** Class. This includes the Class and Object classes. Since Class is itself a class and therefore an object it is a subclass of Object. (see figure 1.4) Note that there is one instance of Class for each of the classes in the figure.

▼ Metaclasses and the ‘class object’ concept are new to Grunt-91. They are not yet well-developed.

*Figure 2.4*



▲ There should be some way to dynamically create objects. At present all storage allocation for objects is performed by the compiler. As observed in [1], because there is one metaclass which is shared by all of the classes the instantiation message would have to be the same for all classes. This is unfortunate only if initialization is done at the same time. Frankly, we do not see that storage allocation and object initialization are inseparable and do not find this particularly objectionable—at least not more so than having a separate metaclass for every class.

A ficticious example of how such a dynamic storage allocator might be implemented:

```
METHOD new [Class Object ||
  (objectMemoryHeap allocateBytes: instanceDataSize ) objectClass: self
  ].
```

New objects would be instantiated by sending new to an object class:

```
(Complex new) (:x:y: 2, 3) (dispose)
```

▲ Of course, with the possibility of incurring run-time memory allocation errors an exception handling mechanism would be useful

## Program

A Grunt program consists of a list of class declarations:

```
program:
      classDeclList;
```

## Declaration

The grammar governing class declaration is as follows:

```
classDeclList:
      classDeclList classDecl;
      ;

classDecl:
      CLASS identifier superClass PARENT class
      | compoundVariableDeclList |
      compoundVariableDeclList
      ENDCLASS;

class:
      identifier;
      METHOD;
```

A class is uniquely identified by the identifier called the class name. Once a class is declared it is accessible from anywhere on in the program. The specified superclass must have been previously declared. Although the keyword METHOD is intended to suggest the existence of a distinct class of method objects, it does not actually define a class unless followed by an initializer in variable declarations.

▼ The implied-superclass declaration from Classic Grunt is no longer allowed—its utility was questionable, at least. Nor is the surrounding class-within-a-class declaration permitted anymore.

▲ The latter syntactical construction might possibly be used in future to introduce subclasses which have more knowledge of their superclass

▲ Not separating declarations from definitions was a mistake which makes parsing more difficult. Worse, it makes it impossible to write a class library without exposing the enitre implementation.

Grunt-91 has name equivalence [10, p. 356] of classes. An exception is block classes which cannot be named and must therefore rely on structural equivalence.

▲ The lack of block class names produces other problems as well

A class is declared when its instance variables have been declared, which are enclosed by the two vertical bars. The following set of declarations are those of the class variables:

```
compoundVariableDeclList:
      compoundVariableDeclList compoundVariableDecl;
      ;

compoundVariableDecl:
      class variableDeclList . ;

variableDeclList:
      variableDeclList , initializedVariableDecl;
      initializedVariableDecl;

initializedVariableDecl:
      variableDecl;
      variableDecl literal;

variableDecl:
      identifier;

variable:
      identifier;
```

A variable is declared by specifying its object class and name. When the METHOD keyword is used the class is defined by the initializer, which must be present. The class of an instance variable must have been previously declared.

A **compound declaration** declares multiple variables of the same object class with a single naming of the class by separating the names with a comma. An **initialized declaration** may be used to give an initial value to an object of some of the system classes. The initialization is, in effect, performed at compile-time and does not involve the sending of messages.

▼ Compound and initialized declarations are new to Grunt-91

For example:

```
| Integer i1, i2 0.  Boolean predicate false.  Character x, y $$n, z.  Integer m |
```

The fact that a class is not yet declared at the time of the declaration of its instance variables implies that the defining class may not be used as the class of one of its instance variables (this is a degenerate case which we call *instance recursion*).

Consider, for example, the declaration of a subclass of Integer called Complex whose instances consist of one inherited integer (its real part) and an instanced integer (its imaginary part).

```
CLASS Complex PARENT Integer
| Integer im 0. |

  METHOD re [Complex Integer || super].
  METHOD im [Complex Integer || im].

  ENDCLASS
```

(A full definition of the Complex class is given in Appendix A.) Class names must hence be unique within a program. By convention they begin with an uppercase, and variable names begin with a lowercase character.

As it is, Complex does not yet implement many methods so most messages sent to its instances will be deferred to its superclass Integer. For example, if z is an instance of Complex, the result of

```
z + 2
```

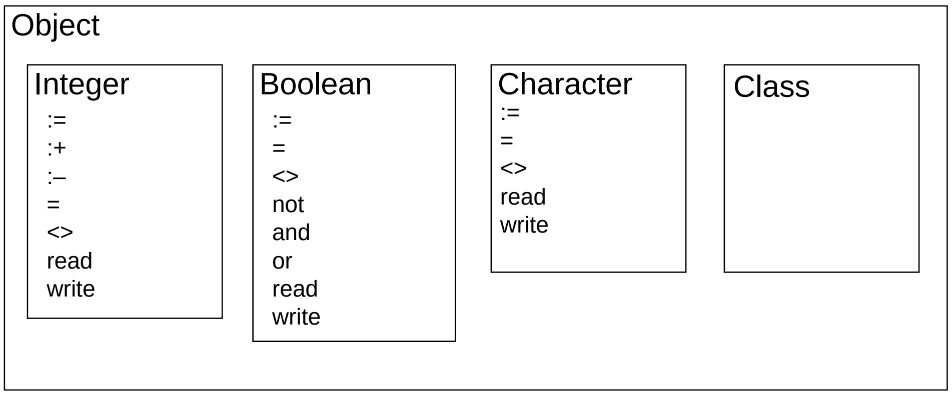
is an Integer object with the value of the real component of z plus two.

▲ This example is at the same time a good argument for having a mechanism like copy constructors C++

## System classes

Because new classes must always succeed from superclasses it follows that there must already be classes from which they can be defined. These classes must also have some predefined methods because likewise new methods are described in terms of other methods. The Grunt system provides five such **system classes**, Object, Integer, Boolean, Character and Class. The Object class serves as a superclass to all other classes. Instances of Class are generated by the run-time environment when the program is started. It is probably not useful for a program to instantiate Class directly. The system class hierarchy is illustrated in the following figure:

*Figure 1.5*



The messages that are understood by their instances are listed in the class boxes. 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, the **system methods**, are shown below in pseudo-Grunt format. (Method declaration is explained in the following section.) The system methods are the only ones which cannot be expressed in Grunt itself but must be implemented directly in the compiler's target language. The declarations are useful because they define proper usage.

```
"
  System class pseudo-definitions

  Imaginary definitions serve to illustrate primitive methods, and
  make use of implicit machine classes
"

CLASS Object
  | Class @objectClass. |
  ENDCLASS
```

Note that in Grunt-91 each object contains a reference to its object class, which is implied by the ficticious @ pointer operator.

```
CLASS Integer
  | MachineInteger i |
  METHOD := [Integer self Integer value || ].
  METHOD :+ [Integer self Integer addend || ].
  METHOD :– [Integer self Integer subend || ].
  METHOD = [Integer Boolean Integer comp || ].
  METHOD <> [Integer Boolean Integer comp || ].
  METHOD read [Integer self || ].
  METHOD write [Integer self || ].
  ENDCLASS
```

```
CLASS Character
  | MachineCharacter c |
  METHOD := [Character self Character value || ].
  METHOD = [Character Boolean Character comp || ].
  METHOD <> [Character Boolean Character comp || ].
  METHOD read [Character self || ].
  METHOD write [Character self || ].
  ENDCLASS

CLASS Boolean
  | MachineBoolean b |
  METHOD := [Boolean self Boolean value || ].
  METHOD = [Boolean Boolean Boolean comp || ].
  METHOD <> [Boolean Boolean comp || ].
  METHOD not [Boolean Boolean || ].
  METHOD and [Boolean Boolean Boolean op || ].
  METHOD or [Boolean Boolean Boolean op || ].
  METHOD read [Boolean self || ].
  METHOD write [Boolean self || ].
  ENDCLASS
```

# 3

# Methods

Methods are a class's way of responding to messages received by one of its instances. A method can effect a response to a message by modifying the receiver of the message in some way and by returning a new *result object*. Methods are the only way through which instances of other classes can gain access to the object.

Practically, a method consists of some program code in the form of a *block; object* which is designed to affect the receiver and result objects in some predetermined way. Methods are merely named blocks which have been associated with a certain class by declaring them as class variables. In fact, the block objects themselves exist quite well independently of a class, and blocks that are not methods may be useful.

▼ In Grunt-91 there is almost no separate method concept. The necessity of blocks, i.e. bits of code that could take arguments and return a result, was already noted in the Classic Grunt language report. Their similarity to methods led to the generalization of methods in terms of blocks. Since methods are block objects of a class, the introduction of class variables was logical. Making classes into objects parallelized class and instance variable objects into a single concept.

## Blocks

New methods are introduced to classes by defining them as variables of the defining class. A method declaration acts to associate a piece of code, called a **block**, with a method name in the defining class. Because methods are no more than named blocks, a block can equally well be sent as a message to a receiver.

▼ Blocks are new to Grunt-91

Although blocks are regarded as if they were regular objects, the classes of which they are instances are unnamed. This means that there is no way to declare a variable block object other than as a method, i.e., using the keyword METHOD as if it were a class name and giving a literal block as an initializer.

The syntax for specifying block literals is:

```
block:
      [ class resultClass optArgumentDeclList
                | compoundVariableDeclList |
                expression
                ] ;

resultClass:
      class;
      self;
      GENERIC;
```

The first class specifies the kind of receiver objects that the block can operate on. Naming an explicit result class defines the class of a result object which is returned by the block when it is invoked. The keywords self and GENERIC introduce certain varieties of blocks called *self-result* and *generic* blocks, explained in more detail further on.

A block may be made to take a number of **arguments**:

```
optArgumentDeclList:
      argumentDeclList;
      ;

argumentDeclList:
      argumentDeclList , argumentDecl;
      argumentDecl;

argumentDecl:
      argumentClass identifier;

argumentClass:
      class;
      GENERIC;
```

A block is considered to be declared once its arguments have been read.

▲ The confusing sequence of class names required in block declarations is not attractive.

Blocks belong to different classes depending on their receiver and argument classes, which must have been previously declared. This means that it is not possible to substitute or assign blocks to each other unless they are compatible in these classes.

Blocks may declare **local variables**, which are created whenever it is invoked and destroyed upon its termination. (The syntax of variable declaration was already given in the previous chapter.)

Variable names must be unique within a block. This means that the declaration of an local variable cannot hide that of an argument variable or another local. Since the instance and class variables of the receiver are accessible to the block, nor may these be hidden by a variable declaration in a block.

The value of the result object is determined at run-time by evaluating the *block expression*. It must be of the same class as the explicit result class, or in case of a self-result method, of the receiver (self) class. Expressions are explained later on.

## *Methods*

As stated previously, a method is a declaration of a block literal as a class variable. For example, within the definition of Complex, the following

```
METHOD im [Complex Integer || im].
```

declares a new method of Complex named im, which creates and returns an Integer result object. Note that the receiver class must be specified in the block, even though it is already known from the fact that it is the defining class. Declaring methods with receiver classes other than the defining class serves no purpose and is not allowed.

Since a method is declared once its initializer is read, and the block is declared after its arguments are read, the method may be used inside the block, allowing the writing of recursive methods.

▼ It was not previously possible to write recursive methods

Messages are identified by their message name, and are not distinguished from their block classes. Since variable names must be unique within the class this implies that Grunt does not permit polymorphic messages. In case a method with the same name was declared in one of the superclasses, the method is said to be *overriding*. The block of an overriding methods must be of the same class as that in its superclass counterparts.

There are some conventions governing the names given to methods. The first alphabetic character in a message name should be a lower-case character (e.g., im). Self-result methods should have message names that begin with a colon (such as :=). Messages that take arguments should end with a colon (im:), except if the last character is already a symbol (+). Messages taking multiple arguments should have one short label and colon for each expected argument (im:re:).

### *self-result methods*

For some methods the creation of a new, distinct result object is not useful and imposes an unnecessary overhead. For example, consider the following assignment method for Complex:

```
METHOD = [Complex Complex Complex new ||
  super := (new re).
  im := (new im).
  self                                         " inefficient "
  ].
```

This assignment works by individually copying each of the argument's instance variables to their corresponding receiver variables, and then copying the receiver into a new result object. This result object conveys no additional information as it is a copy of the receiver.

In cases such as this when the receiver object is returned as the result, a **self-result** block is more efficient. The result of such a method is not returned in a new object but in the receiver itself.

A self-result method is specified by using the keyword self instead of naming a result class in the method declaration. The above assignment expressed as a self-result method:

```
METHOD := [Complex self Complex new || new ].
```

As the declaration suggests the receiver object itself is the result of the message. This means that new is stored directly into the receiver.

Consider the self-result version of re:im::

```
METHOD :re:im: [Complex self Integer newx, Integer newy ||
  super := newx.
  y := newy.
  self
  ].
```

Even though the block expression looks the same, this method is more efficient because it does not cause an unnecessary result object to be created. In general, methods that do not *create* information outside of the receiver are candidates for self-result methods.

It should be remarked that the meaning of sending self-result messages is still equivalent to that of an ordinary message. That is, a self-result message may be used anywhere another message would. However, the next section shows that self-result messages can be used in instances where others may not.

### *Generic methods*

A **generic method** is one which takes arguments and returns a result of a single undetermined object class.† A generic block is declared by using the GENERIC keyword in place of the result class:

```
METHOD if:then:else: [Boolean GENERIC GENERIC then GENERIC else ||]
```

The result of an invocation of a generic method must be one of the generic arguments. Since the actual object class of a generic argument cannot be deduced from within the block, they cannot be sent messages.

## Expressions

An **expression** denotes a combination of objects in various ways in order to achieve a desired effect. In blocks, an expression evaluates into the **result object** which is returned by the block.

```
expression:
      expression . object;
      object;
```

† The Grunt generic concept, though similar, differs from that in Ada in that the method remains generic at run-time. Ada creates a different version of a generic procedure for each type that it is used with

Expressions can be concatenated by separating them by a period. The expression is evaluated left to right, and the evaluation of the entire expression is the object evaluated by the rightmost expression.

For example, from the above definition of the Complex class method :re:im::

```
super := nre. im := nim. self
```

represents the Complex receiver object after the two Integer assignments have been performed.

Objects can be formed from other objects by sending them messages:

```
object:
      simpleObject messageChain message;
      simpleObject;

message:
      messageName optParameterList;
```

## *Message chaining*

A message starts with the method name, followed by any parameters it takes separated by commas. If the message is implemented as a self-result method, it may be enclosed in parentheses and followed by another message, which is then directed at the same receiver object. This construction is called a **message chain**:

```
messageChain:
      messageChain ( message ) ;
      ;
```

For example, the expression

```
z (:write) (:re:im: 2, 3) :write
```

sends the messages :write, :re:im: 2, 3 and again :write to the same object z. The messages enclosed in parentheses must all be self-result messages. The last message in the chain should not be parenthesized. This is different from

```
((z :write) :re:im: 2, 3) :write
```

which sends a message :write to z, creates a new object, sends it :re:im: 2, 3, creates a new object and sends it the message :write. Note how the parentheses suggest a different object.

One might wonder why message chains with other than self-result messages are not allowed. If such an expression were permitted the result object returned by the non-self-result message would be lost, such as in the expression

```
z (+ 6) :write                              " illegal "
```

In such a construct the result object answered by the message + 6 would be inaccessible. This agrees with an observation made previously, that messages that do not create new information should probably be implemented as self-result methods.

▼ In Classic Grunt there was no way to send the same object more than one message other than by respecifying it. Naturally this did not work for literal objects.

```
simpleObject:
      ( expression );
      variable;
      literal;
      self;
      super;

literal:
      integer;
      character;
      boolean;
      block;
```

When an expression which does not already evaluate as a temporary object is enclosed within parentheses, it is made into a new temporary object with the same value. For example, z and (z) and (z) are each different objects. This is especially useful because it makes it unnecessary to write two versions of a method, one that is self-result, and another that returns a new object. For example, of

```
METHOD :+ [Complex self Complex addend ||
  " adds the argument into the receiver"
  ].
METHOD + [Complex Complex Complex addend ||
  " adds the argument and the receiver into a new object "
  ].
```

the second is redundant since the behavior of z + w is equivalent to (z) :+ w.

## *Pseudo-objects*

The keywords self and super signify so-called **pseudo-objects**. Within a block, the psuedo-object self refers to the receiver object of the block.

▼ In Grunt-91 self really *does* refer to the receiver object. In Classic Grunt objects did not retain any information about their object classes at run-time, so that a method always perceived the receiver object as if it were of its own defining class. (‘Method’ dispatches were like nonvirtual member function calls in C++. See, for instance, the edifying remark in [9, pp. 208-209].)

The pseudo-object super refers to the receiver in the superclass of the defining class. It is the only way through which a block can gain (indirect) access to the instance variables of the defining class's superclass. So we would write a method to return the real part of a Complex by writing

```
METHOD x [Complex Integer || super].
```

## Arguments

```
optParameterList:
      parameterList;
      ;

parameterList:
      parameterList , simpleObject;
      simpleObject;
```

Often a method uses other objects which are passed to it together with the message. Such objects are called **arguments** to the method. For example, a method which assigns a value to a Complex based on its real and imaginary components may be defined as

```
METHOD :re:im: [Complex self Integer nre, Integer nim ||
  super := nre.
  im := nim.
  self
  ].
```

This defines a message :re:im: taking two Integer parameters. Such a method is invoked as its declaration suggests, for example, by sending the message

```
z :re:im: 2, 3
```

## Overriding

We want to redefine addition for complex numbers so that :+ will take a Complex argument:

```
METHOD :+ [Complex self Complex addend ||
  super :+ (addend x).
  y :+ (addend y).
```

```
        self
        ].
```

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.

# 4

# Implementation

This chapter describes a generally platform-independent implementation of a Grunt-91 compiler and run-time environment, as well as a specific implementation under the UNIX operating system running on a Sun-4 SPARCstation and under the Apple Macintosh's Macintosh Programmer's Workshop.

Since the compiler was written in portable C with the aid of a LEX-compatible lexical analyzer-generator and the YACC LALR(1) parser-generator, we expect to be able to quickly port it to other environments—particularly, the Macintosh Programmer's Workshop. In addition, because we have tried to design a very general intermediate representation, retargeting the compiler (possibly for the Qbe*rt hypercube processor [7]) should be straightforward.

It is somewhat interesting to note that while experimenting with several different forms of expression syntax, most of them (including the one from Classic Grunt) usually gave rise to a number of shift/reduce and an occasional reduce/reduce conflict. We had not anticipated conflicts arising from error nonterminals. Although certainly at least the shift/reduce conflicts could be quite favorably resolved, the resulting expressions easily became completely unreadable. The form we at last decided on, and in fact the entire Grunt-91 syntax, is completely LR(1) and does not contain any grammar conflicts at all. We do not feel the language to have been constrained by this decision. In fact, the LR(1) syntax was at the same time the most human-intuitive.

## Compiler Implementation

The implementation of the Grunt-91 compiler actually consists of three separate programs, each using a different code generator. One of these is intended for compiler development and generates a human-readable English ‘object code’ for use in debugging. The second generates code for the Motorola MC68000 family of processors with an optimization option for the 68020/030. The code has been fine-tuned to assemble under the Macintosh Programmer's Workshop assembler and environment. Although it would have been only slightly harder to generate the machine code directly, this would have made the output more platform-dependent than we desired. As an experiment in compiler retargeting, the third compiler generates (tentative) assembler code for the Motorola MC88100 RISC processor. The code from this compiler could not be completely verified because there was no machine available to us to run it on.

The compilers can be built easily using the `make` command. The target language of the compiler is specified by setting the `makefile` variable `TARGET` to one of `debug`, `68000`, or `88100`, or by specifying it directly from the command line:

```
make TARGET=target
```

A new target architecture can be readily incorporated by writing a code generator module in a file named `c`*`target`*`.c`, where *target* is the name of the target architecture, and rebuilding the compiler.

A compiler is invoked from the UNIX or MPW command line with:

```
grunt [-v] [-c020] [-o file] [file]
```

where the Grunt source is read from standard input if no input file is specified, and the object code is written to standard output if no explicit output file is specified. The options recognized are:

| | |
|---|---|
| `-o` *`file`* | generate object code into named file |
| `-v` | verbose. Generates readable comments into the object code |
| `-c020` | (68000 compiler only) Generate 68020 code |

Generic blocks are not implemented by the compiler.

## *Observations*

As stated, the current implementation was written in C with LEX- and YACC-compatible parser building tools. This is a much happier situation than using the ad hoc tools which were provided in the original Vertalerbouw course for the implementation of Classic Grunt and which now forced us to restart the compiler-building process from scratch.

We switched to using the FLEX lexical analyzer-generator when we found that LEX apparently could not handle our perhaps somewhat unusual requirement of using many symbols in identifiers. Unfortunately, FLEX (adhering more closely to the LEX specification than LEX) does not support the undocumented `yylineno` variable which we had used to good advantage in reporting the location of Grunt program errors. We have found it impractical to attempt to patch our own `yylineno` into the generated lexical analyzer. Nor were we successful at adding a pattern-matching rule to provide this information.

We found that YACC does not support attributes as well as we would have liked. In particular, inherited attributes are not effectively implemented, although we are glad the authors left a back door so that they could be used. Other than this we have had little trouble using it. Use of the explicit `error` nonterminal, we find, is a clean and consistent way of expressing permissible (recoverable) parsing errors.

Also we would like to express our continued amazement at the apparent dearth of ANSI C compilers. K&R C has again effectively reminded us of the benefits of function prototypes. The `dbx` source-level debugger was invaluable in tracking down errors.

Interesting are comparisons between the different compilers. The number of lines of the assembly object code (with the `-v` option off, roughly equal to the number of emitted instructions) for the Complex example in Appendix A:

| | | |
|---|---|---|
| 68020 | 233 | lines |
| 68000 | 279 | |
| 88100 | 387 | |

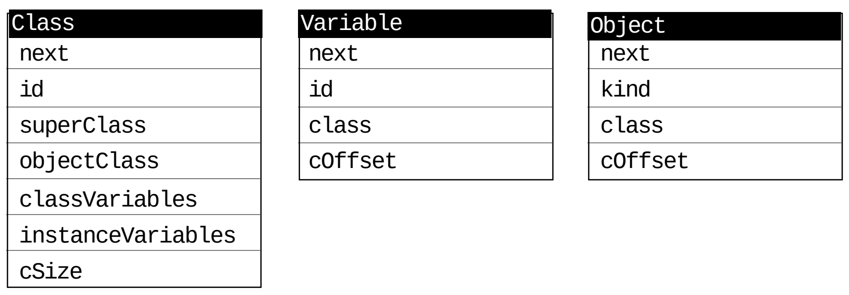
Although we cannot comment on the code efficiency or degree of optimization that the respective compilers achieved, we found that the code for the RISC processor was easier to generate due to the small number of addressing modes and greater orthogonality of the instruction set. Another way of saying the same is that the CISC processor allowed more effective code to be generated using its more extensive facilities.

## **Syntax Tree**

The first half of what we consider the intermediate representation of a Grunt program is the syntax tree which is built during parsing. This tree is accessible to the code generator module to retrieve any information that it requires. The syntax tree has shown itself to be a powerful and flexible way of directing code generation because it does not limit the information that is available any more than inherent in the language.

The syntax tree consists of the following three types of nodes:

*Figure 4.1*

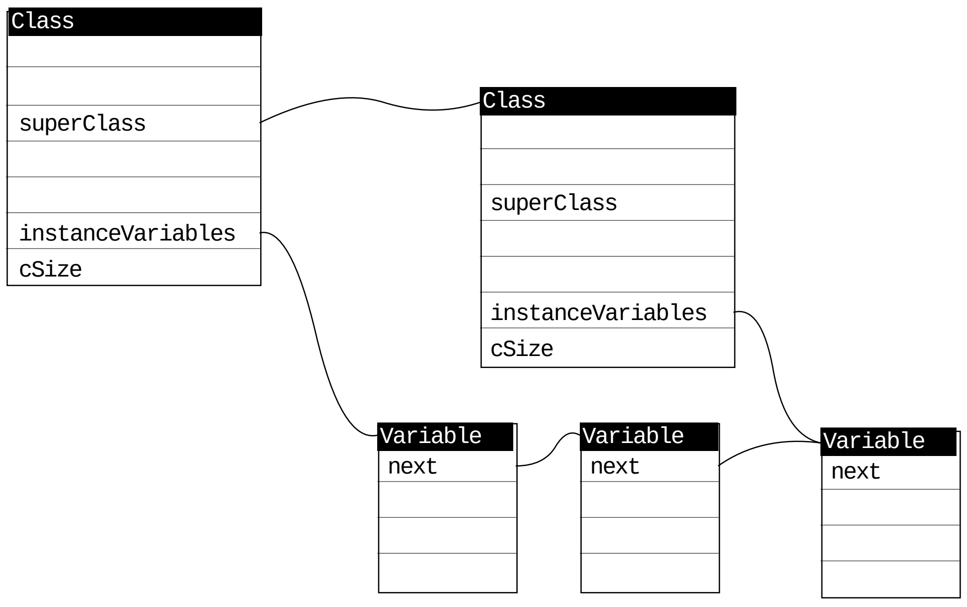


In all three types, `next` is a link to another node of the same kind in a linked list, or `NULL` at the end of a list. New nodes are added in front of a list. The type of list in which the node appears is context-dependent. `id` is a pointer to the symbol table entry of the node's identifier.

## *Classes*

Class nodes define object classes. `superClass` and `objectClass` point to the class nodes of their super- and object classes. `classVariables` and `instanceVariables` are linked lists of variables. `cSize` is the sum of instance sizes of the instance variable classes. The instance variable list of a superclass is shared by any of its subclasses, so the nodes are not duplicated:

*Figure 4.2*



For a variable node `class` is its object class. `cOffset` is the sum of instance sizes of the classes of all variables further in the list and not including that node itself (i.e., those added to the list before it).

## *Objects*

An object node specifies the type and the run-time location of an object. `kind` specifies one of the following six types of object:

- Receiver (either self or super)
- Receiver variable
- Class variable
- Local
- Argument
- Temporary

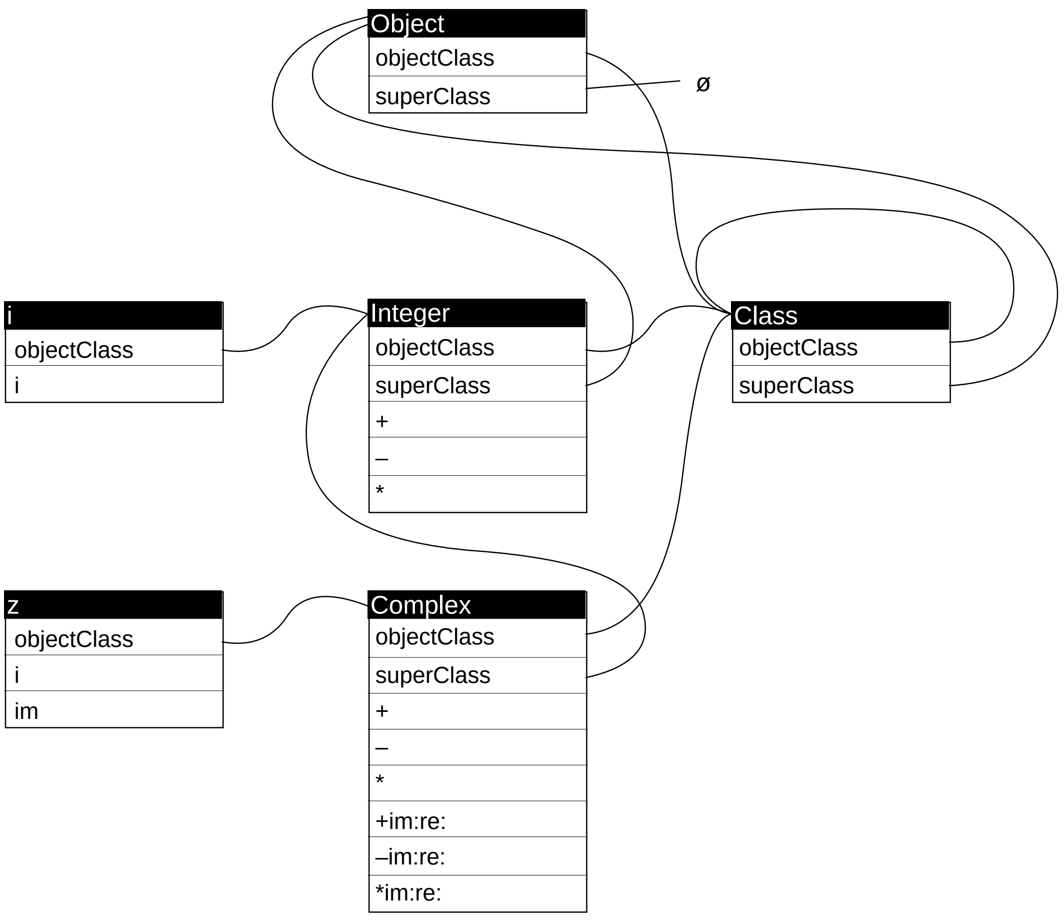
This and the offset within the containing object, constructed with information supplied by the code generator, completely specifies the location of an object.

## Code Generation

The second part of code generation is in explicit procedure calls by the parser to the generator.

▲ In places where we do not translate the parse into construction of a syntax tree but use sequenced calls to the generator (for instance, expression evaluation), coordinating parsing with code generation has turned out to become more difficult.

## Run-time Environment



## *System classes*

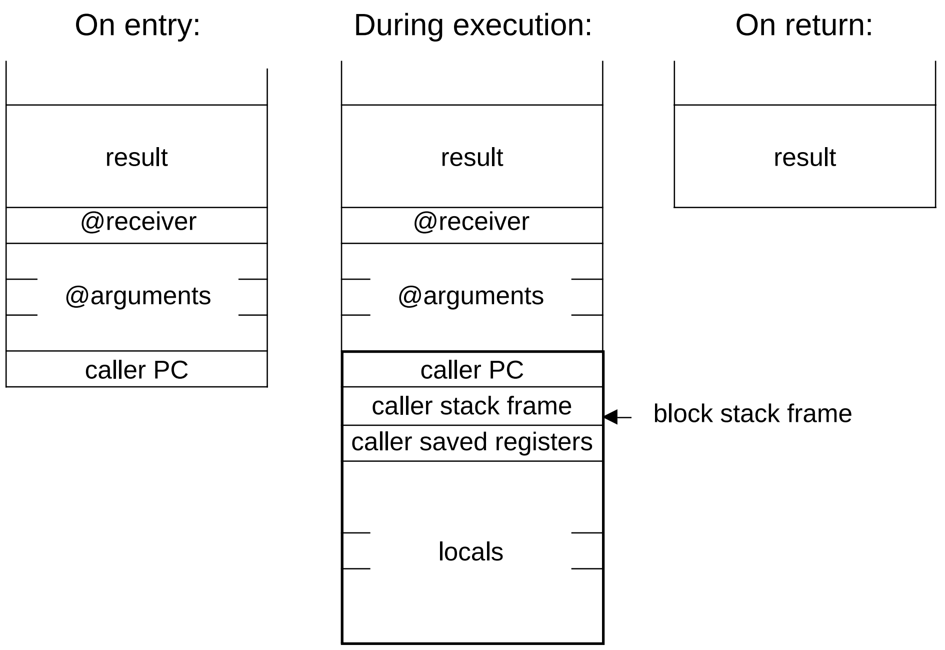
System classes are automatically declared by the compiler before any of the user classes. Instance variables of the system classes typically consist of data fields representing the actual ‘contents’ of the object, and a pointer to their object class. Because Grunt-91 does not have a class representing ‘meaningless’ physical memory, nor an object pointer concept, the instance variables of the system classes do not have object classes. In the implementation of the compiler it is pleasant not to have to treat such instance variables too differently, so rather than nothing at all they are given **void classes**, which are not real classes at all but placeholders for class-like data. Void instance variables of system classes are not identified so that they cannot be referred to in a Grunt program. Note that void classes are a *compiler implementation* and not a *language* concept.

System classes are declared and generated (which involves bootstrapping the run-time environment) by the `DeclareSystem` routine in the `system.c` module. After this initialization there is no distinction between system and user classes.

## *Message sending*

Because methods are just block class variables, and blocks are represented by the address of their entry point, method dispatching involves the simple retrieval of this address from the object's class object. Methods, like other variables, are located by an offset into the class object and there is no ‘method ID’. Subclasses inherit their superclass's class variables by containing explicit copies of those variables. A Grunt program with 100 classes and 100 methods each therefore requires approximately 40kB of storage to support the inheritance mechanism. There are means to reduce this at the cost of a more complicated and slower method dispatch.

By improving the way in which messages are parsed, the need for an explicit argument size parameter, as was used in the implementation of Classic Grunt, has been obviated. In particular, the result object is now created and the argument list parsed before any argument pointers are pushed onto the stack. In fact, from a code generation standpoint, the entire message send is atomic.



A message is directed at and has as parameters specific *objects*, not *values*, so these therefore must be passed by reference.

## *Temporary objects*

Temporary objects are objects that are created implicitly by the compiler to support the sending of a single message. They arise when literals are used, when expressions are enclosed in parentheses, and when expressions are combined in expression lists.

Although the Grunt-91 language specifies that a temporary object cannot exist after the expression in which it occurs is evaluated, this is still somewhat academic. Because there is no implicit object disposal mechanism a program has no way of knowing whether a temporary object still exists or not.

Still, the implementation creates and removes temporary expression result objects in line with its evaluation. Literal temporaries are created before the message send and removed immediately afterwards. Note that this implies that expressions used as arguments to messages must be evaluated before the argument pointers are pushed onto the stack.

# A

# Examples

This chapter gives a concise example of Grunt-91's new features, and gives an implementation of a class of complex numbers. The listings of code generated from this class are available from the authors.

## *Factorial*

First of all the factorial computation example from the Classic Grunt report, which illustrated the need for method recursion, conditional evaluation and blocks:

```
METHOD ! [Integer self ||
  self (
    (self = 0) ifTrue:False:
      [Integer self || 1],
      [Integer self || (self – 1) (!) * self ]
    )
  ].
```

! operates by sending self one of two blocks and answering the recevier as its result. Which block is sent depends on whether self equals 0. If it does, the block returns 1 when sent to an Integer. Otherwise, the block sends a new temporary object (self – 1) the messages ! and * self in a message chain and returns this as its result.

Compare this with a Pascal version of the same:

```
FUNCTION Factorial(i: Integer): Integer;
BEGIN
  IF i = 0 THEN
    Factorial := 1
  ELSE
    Factorial := Factorial(i - 1) * i;
  END;
```

## *Complex class*

```
"
    Complex.grunt

    complex number class

    Version 1.0      22/04/91

    (c) Copyright 1991 by: Ben Hekster
"

"
    Complex class definition

    Inherited:       real part
    Instanced:       imaginary part
"
```

```
CLASS Complex PARENT Integer
| Integer im. |

    "   re/re:
        Returns/sets the real part
    "
    METHOD re [ Complex Integer || super ].
    METHOD re: [ Complex self Integer new || super := new. self ].

    "   im/im:
        Returns/sets the Complex's vertical coordinate
    "
    METHOD im [ Complex Integer || im ].
    METHOD im: [ Complex self Integer new || im := new. self ].

    "   :0
        :1
        Reset a complex number
    "
    METHOD :0 [ Complex self || super := 0. im := 0. self ].
    METHOD :1 [ Complex self || super := 1. im := 0. self ].

    "   :=
        :re:im:
        Assigns a complex number to the Complex
    "
    METHOD := [ Complex self Complex new || new ].
    METHOD :re:im: [
        Complex self Integer nre, Integer nim || super := nre.
        im := nim. self ].

    "   :+
        Adds a complex number to the Complex
    "
    METHOD :+ [ Complex self Complex addend ||
        super :+ (addend re). im :+ (addend im). self
        ].
    METHOD :+re:im: [ Complex self Integer nre, Integer nim ||
        super :+ nre. im :+ nim. self
        ].

    "   :read
        Read a complex number from input
    "
    METHOD :read [ Complex self ||
        super :read. im :read. self
        ].

    "   :write
        Write a complex number to output
    "
    METHOD :write [ Complex self ||
        super :write. $, :write. im :write. self
        ].

    ENDCLASS
```

# B

# References

[1] Adele Goldberg and 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] Michael C. Storrie-Lombardi, *Smalltalk/V Mac: A New Standard in Object-Oriented Programming,* MacTech Quarterly, volume 1 number 2, Summer 1989

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

[7] Ben Hekster. Peter F.A. Middelhoek, and Robert Remmers, *Qbe*rt: a Massively Parallel Hypercube Processor,* course report, to be published

[8] Bell Laboratories, *UNIX™ Time-Sharing System, Volume 2: Programmer's Manual,* seventh edition, Holt, Rinehart and Winston

[9] Margaret A. Ellis and Bjarne Stroustroup, *The Annotated C++ Reference Manual,* Addison-Wesley

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

# Index