The Green Language

José de Oliveira Guimarães
Departamento de Computação
UFSCar
São Carlos - SP
Brasil
e-mail: jose@dc.ufscar.br

22 de maio de 2007

Sumário

Capítulo 1

Introduction

This Report defines the Green Language, an object-oriented language being designed at the Computer Science Department of the Federal University of São Carlos (*Universidade Federal de São Carlos - UFSCar*). Language Green separates the subtype from the subclass hierarchy and supports garbage collection, classes as first-class objects, parameterized classes, introspective reflection and a kind of run-time metaobjects called shells.

Green will support compile and link-time metaobjects and will have a special module system. These features have not been completely defined and are not discussed in this report.

This is not an introduction to object-oriented concepts. We assume the reader knows at least one object-oriented language well.

Capítulo 2

Basic Elements

2.1 Comments

In Green, anything between /* and */ is a comment. Nested comments are allowed. So the line,

```
i = 10; /* comment /* i = 5; old code */ still a comment */
```

is legal and contains one assignment.

There is another way to specify a comment: anything after // till the end of the line is a comment.

2.2 Basic Types

The language basic types are shown in the table below together with the operators they support. The comparison operators are supported by all types.

basic type	operators
char	`++ --`
boolean	`and or not xor`
byte	`+ - * / % & | ^ ~ ++ -- << >>`
integer	`+ - * / % & | ^ ~ ++ -- << >>`
long	`+ - * / % & | ^ ~ ++ -- << >>`
real	`+ - * / ++ --`
double	`+ - * / ++ --`

`+`, `-`, `*`, and `/` are the arithmetic operators. The remainder of division of `p` by `t` is given by "`p%t`". Operators "`&`", "`|`", and "`^`" are the bitwise "and", "or", and "xor" (exclusive or). "`~`" is the bit to bit complement.

Operators `++` and `--` are only applied to variables and are prefixed. They increase (`++`) or decrease (`--`) their operands by one and do not return a value, unlike C++.

The operators `<<` and `>>` are right and left shift of bits. In an expression

```
k << n
```

`k` may be `byte`, `integer`, or `long` and `n` may be `byte` or `integer`.

The Green compiler translates source code into C. The table below shows the mapping between Green and C basic types.

Green	C
`char`	`signed char`
`boolean`	`char`
`byte`	`unsigned char`
`integer`	`int`
`long`	`long`
`real`	`float`
`double`	`double`

So, the semantics of the Green basic types depends on the semantics of the basic types in the C compiler and machine used. We hope to fix the size of `integer` in 32 bits and introduce 16-bit `short` integers. Chapter 8 discusses other features of basic types.

Table 2.1 shows the precedence of the operators. Operators in the same box have the same precedence. Unary operators and the assignment operator are right associative. The operators

```
<<   >>   <   <=   >   >=   ==   <>
```

are neither left or right associative. That means expressions like

```
(a << 5 << 1) == b == c
```

are illegal. Every other operator is left associative. The higher in the table, the higher the precedence. Any similarities between this table and one of Stroustrup book [17] is not a coincidence, although the operator precedence order in Green is rather different from C++.

Operators `and` and `or` finish their execution as soon as possible. Then, in

```
ok = false and good();
ok = true or good();
```

the `good` method would not be called.

2.3 Literal Values

A character constant should be enclosed between `'` as in language C. All escape characters of C are valid in Green.

The `boolean` type has two pre-defined values: `true` and `false`. These are constants that can be cast to `1` and `0`, respectively.

A byte value is a literal integer with a postfixed "`b`". For example,

```
2b, 32b, 255b
```

are byte values.

Any literal number without a floating point or exponent is an integer literal number that can be postfixed with an `i`: `3200i`, `-1i`.

`long` values must be postfixed with an upper case `L`:

```
3L,   3000000L
```

Any literal number with a floating point or an exponent is a `real` number. Example:

```
1.1   320E+5   1e-10   1r
```

A real number may have a trailing "`r`" as in `1r` above.

A double real value must have a "`d`" at its end:

```
1d   2.1e-10d   445.076d
```

The rules for valid numbers (`byte`, `integer`, `long`, `real` and `double`) are the rules of the C compiler used.

`()`	method call
`[]`	subscripting
`.`	method/variable selection
`~`	bitwise complement
`+`	unary plus
`-`	unary minus
`not`	logical not
`++`	increment
`--`	decrement
`<<`	shift left
`>>`	shift right
`&`	bitwise and
`^`	bitwise xor
`|`	bitwise or
`/`	divide
`&`	multiply
`%`	remainder
`+`	binary plus
`-`	binary minus
`==`	equal
`<>`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`and`	logical and
`xor`	exclusive or
`or`	logical or
`=`	assignment

Figura 2.1: Operator precedence table

No automatic conversion is made among values of the different basic types. Any conversion must be explicitly made. For example, to assign a real variable `r` to an integer variable `i` one should write:

```
i = integer.cast(r);
```

This subject is further discussed in Chapter 8.

2.4 Identifiers

An identifier is a sequence of one letter followed by any number of letters, digits, and underscore ("_"). There is no limit for the size of an identifier, although there may be problems when translating long names into C source code. Upper and lower case are considered different. We expect that the Green compiler will issue a warning if two identifiers differing only in the case of the letters are used in the same scope.

2.5 Assignments

Green uses `=` for assignments which are considered expressions. Then the code

```
a = b = 1;
c = (i = getInt()) + 4;
while (ch = readCh()) <> '\0' do
  ;
```

is legal.

2.6 Control Statements

The `if` statement of Green has two legal forms:

```
if expr then stat-list endif
```

or

```
if expr then stat-list else stat-list endif
```

For example,

```
if i > 0 and a < b
then
  k = 0;
else
  super.put(k);
endif
```

is a valid `if` statement.

There is also a `case` command, as shown below:

```
case i of       // i = case expression
  0 :           // label expression
    readData();
  1, 2, 3 :     // label expressions
    writeDate();
```

```
  4 :
    begin
    k = read();
    write(k + 1);
    end
  end // case
```

After the keyword `case` there may appear an expression that we will call "case expression". The constant expression of each option (like `0`, `1`, `2`, `3`, `4`) will be called "label expression". The case expression should result in:

- an object of any type.[1] Then the label expressions should be class names;
- a value of type `char`, `boolean`, `byte`, `integer`, or `long`. The type of each label expression should be equal to the type of the case expression.

The values "`0`", "`1, 2, 3`", and "`4`" in the `case` example just given are called "*labels*". There may be a label "`otherwise`" chosen when no other label applies.

```
i = 3;
case i of
  1 :
    ++j;
  2 :
    --j;
  otherwise :
    error();
end
```

2.7 Loop Statements

Green supports four different types of loop statements. Since they are very simple and similar to other language constructions, we do not believe they complicate the language in any way.

The `for` statement has the syntax:

```
for Id = expr1 to expr2 do UnStatBlock
```

where `Id` is a `char`, `byte`, `integer`, or `long` variable and `expr1` and `expr2` are expressions of the same type as `Id`. `UnStatBlock` is either a single statement or a block of statements delimited by `begin-end`. Variable `Id` may be declared in the `for` statement:

```
for Id : integer = expr1 to expr2 do UnStatBlock
```

`for` makes `Id` assume the values from `expr1` to `expr2` while iterating through `UnStatBlock`. Therefore, `expr2` should be greater than `expr1`. To `Id` is added `1` in each iteration. If `expr2 < expr1`, the `UnStatBlock` is not executed. The value of `Id` after the `for` is undefined. Of course, `expr1` and `expr2` are evaluated before the first iteration. `expr1` is evaluated first. `Id` should be a local non-array variable.

The `while` statement has the form:

```
while expr do
  UnStatBlock
```

[1] Types and classes are defined elsewhere.

`UnStatBlock` is executed until the boolean expression `expr` becomes `false`. If `expr` is `false` the first time it is evaluated, `UnStatBlock` is never executed.

There is a `repeat-until` statement:

```
repeat
  read(month);
until month >= 1 and month <= 12;
```

The statement

```
repeat
  stat-list
until expr;
```

is defined as:

```
stat-list
while not (expr) do
  stat-list
```

That is, `stat-list` is executed until `expr` is `true` and at least one time.

The statement

```
loop
   stat-list
end
```

is equivalent to

```
while true do
  begin
  stat-list
  end
```

The loop is interrupted when a command `break` is executed. Note that `break` can only be used within the `loop-end` statement. It cannot be inside a `for`, `while`, or `repeat-until` statement that is inside the `loop-end` instruction.

Capítulo 3

Classes and Methods

3.1 Syntax

A class is a type declaration used to create objects of that class. The syntax employed for class declaration is shown in Figure 3.1. A class can declare methods and instance variables, which are called "function members" and "data members" in the C++ jargon.

A class can have a public and a private part, as indicated by "`public:`" and "`private`", respectively. The public part can only declare methods. The private part can declare methods and instance variables and is visible only inside the class code. There may be just only public and one private part and the public should appear before the private part. Before the public part there may be one or more constructors, all of them with the same name `init` but with different number and/or parameter types.

A method is declared as shown in Figure 3.2 where the return value types are optional. `parameter-list` is a sequence of variables with their types, as in Pascal and all parameter passings are by value. The declaration of local variables and instance variables is preceded by the keyword `var`. The declaration of methods is preceded by `proc`. An example of class declaration is in Figure 3.3.

A local variable can be declared in any place a statement can appear using the syntax

```
var k : integer = 0;
```

The scope of the variable is the point of declaration till the end of the method.

In this case the keyword `var` is used to declare a single variable. Then, the code

```
var i : integer = 0, ch : char = '?';
```

in a method body is illegal.

The return value of a method is set by the keyword `return` as in language C. After the command `return` is executed, the method finish its execution and the control is returned to the caller of the

```
class ClassName
    // constructors
  public:
    // methods
  private:
    // methods and instance variables
end
```

Figura 3.1: Syntax of class declaration

```
proc MethodName( parameter-list ) : ReturnTypes
  // Local Variable Declarations
begin
  // method body
end
```

Figura 3.2: Syntax for method declaration

```
class Store
    proc init() begin end
  public:
    proc get() : integer
      begin
      return x;
      end
    proc set( px : integer )
      begin
      x = px;
      end
  private:
    var x : integer;
end
```

Figura 3.3: Class `Store` in Green

method. Even if the method is not supposed to return anything, one can use `return` although without an argument.

Variables and parameters refer to objects and are similar to pointers in Pascal/C. Therefore in the code

```
...
  var a, b : A;
begin
...
b = A.new();
a = b;
...
end.
```

the declaration of `a` and `b` does not lead to the creation of two objects of class `A`. These are created by the method `new`, using dynamic memory allocation:

`b = A.new();`

There is no way to deallocate memory, which is made automatically by the garbage collector.

The statement `a = b` makes `a` and `b` refer to the same object, thus creating an aliasing. Any alteration in the `a` object by means of a message send will be reflected in the object referred to by `b`, since they refer to the same object. A variable of one of the basic types does contain a value — it is not a pointer.

3.2 Message Send

The statement

```
a.set(1)
```

is the sending of the message "`set(1)`" to the object `a`.[1] "`set`" is the message name and `1` is the real parameter.

Suppose `a` was declared as

```
var a : Store;
```

and refers to a `Store` object at run time. Then, the statement "`a.set(1)`" will order the run-time system to search for a method `set` taking one integer parameter in the class of the object `a`, which is `Store`. This method is found and executed. Inside method `set` of `Store` (here on, `Store::set` — see Figure 3.3), the statement

```
x = px
```

means "`a.x = px`" because message `set` was sent to `a`. All references to the instance variables refer to the instance variables of object `a`.

In a message send the value returned by the executed method need not to be used, as in C++. Then, the statement

```
a.get();
```

is legal. Of course, the compiler should issue a warning message.

3.3 self

The keyword `self` is a predefined variable that points to the object that received the message that caused the execution of the method. So, `a.set(1)` caused the execution of `Store::set` and inside this method `self` refer to the same object as `a`. Then, the statement "`x = px`" can be rewritten as "`self.x = px`". Variable `self` cannot be used in the left-hand side of an assignment.

3.4 Method Overloading

Method overloading is supported. A class may define two methods with the same name if they differ in the parameter types or number of parameters. Then the methods

```
proc print();
proc print( width : integer );
proc print( w : integer; y : integer );
proc print( x : Figure );
proc print( x : Circle );
proc print( w : Person );
```

may all belong to the same class. For limitations on the use of overloaded methods, read Section 7.6.

3.5 Assertions

Assertions are expressions that must be true before and after the execution of a method. For example, the method

[1] We will use object "`a`" instead of "object `a` refer to".

```
proc push( x : integer )
  assert
    before not full();
    after  not empty();
  end   // end of the assert clause
  begin
    // push x in the stack
    ...
  end
```

defines pre and postconditions between the `assert` and the first `end` keyword. The pre and postcondition are the expressions after the keywords `before` and `after`. If this method belongs to class `Stack`, the message send to `x` in

```
x = Stack.new();
x.push(5);
```

makes the system test "`not full()`". If this evaluates to `false`, one of two things will occur:

- method
  `proc correctAssertionBefore( mi : MethodInfo )`
  of class `Stack` will be executed, if class `Stack` has such a method. "`mi`" is an object that describes the method that caused the exception — see Chapter 16. This method should be public. Of course, it is inherited by subclasses;
- exception `AssertionBeforeException` will be throw if class `Stack` does not have a method `correctAssertionBefore`.

After method `push` is executed, the expression "`not empty()`" is tested. If this expression evaluates to `false`, one of two things will occur: if `Stack` has a method `correctAssertionAfter`, this is executed. Otherwise, an exception `AssertionAfterException` will be thrown. Note the expressions of parts `before` and `after` may call public (not private) class methods.

Variables can be declared between the before and after clauses. There should be one variable for each `var` keyword and the variable should be initiated in the declaration. As an example, suppose the stack should store only positive numbers and `getSize` is the `Stack` method that returns the number of elements in the stack. The new method `push` could be:

```
proc push( x : integer )
  assert
    before
      x >= 0 and not full();
    var oldSize : integer = getSize();
    after
      not empty() and getSize() == oldSize - 1;
  end

  begin
    // body of push
    ...
    end
```

`oldSize` is like a local variable whose scope is from its declaration to the end of the assert clause. All assert variables are initiated before the method is called. The value returned by a method, if there is only one, is accessed by variable `result`:

```
abstract class Container
  public:
    abstract proc add( x : integer )

    abstract proc get() : integer

    proc empty() : boolean
      begin
      return getSize() == 0;
      end

    proc full() : boolean
      begin
      return getSize() == getMaxSize();
      end

    abstract proc getSize() : integer

    abstract proc getMaxSize() : integer;
end
```

Figura 3.4: Class `Container` for storing objects

```
proc cast( ch : char ) : integer
  assert
    after result >= 0 and result <= 127;
  end
```

3.6 Abstract Classes

A class in which some method bodies may not defined is called *abstract*. The class declaration is preceded by the keyword `abstract` and so are the methods not fully defined. Although the body of a method (its local variables and statements) may not be specified, its header should. As an example, the class `Container` of Figure 3.4 is abstract and the method bodies of `put`, `get`, `getSize`, and `getMaxSize` are missing.

To fully define abstract classes we need to specify some features and restrictions, which is made below.

- constructors, defined in Chapter 4, cannot be abstract;
- some methods and instance variables may be fully defined and the methods may use other class methods;
- an abstract class may not have any abstract method. It still is an abstract class as long as the keyword `abstract` precedes the word `class` in the class declaration;
- constructors may be defined although an object of an abstract class cannot be created — abstract classes are made only to play the role of superclasses and as a design specification.

A variable whose type is an abstract class can be declared and it can refer to objects of subclasses of the abstract class;

- all abstract methods must be in the public or subclass section. See Section 5.4 for the definition of subclass section.

3.7 Why Green Does Not Support Operator Overloading ?

Operator overloading is the use of operators of the basic classes (`+`, `*`, `<=`, `and`, ...) as method names in normal classes. In general this makes the language more complex without adding no power to it. Other reasons for not supporting this feature are discussed next.

When one defines a method

```
proc +(other : T) : T
```

in a class `T`, we assume `+` has the same semantics as the basic-class operator "`+`". That is, "`+`" is free from side effects. The evaluation of "`x + y`" does not change the value of `x` or `y` if they are integers.

It is tempting to introduce a slightly change to the semantics of a method

```
proc +(other : Matrix) : Matrix
```

of class `Matrix` in such a way the return value is `self` then saving the creation of a large `Matrix` object that would be returned by "`+`". Parameter `other` would be added to `self` by `Matrix::x` then producing a side effect. This is the reason why normal classes cannot declare operators as methods.

Besides the side-effect problem (semantics differences), there is another related to this: there is no simple way a method can emulate some operators like `and` and `or` of class boolean. These methods have short-circuit evaluation. This means an expression

```
i < n and v[i] > 0
```

evaluates to `false` if `i >= n`, regardless of the result of `v[i] > 0`.

Suppose operator `and` was added to a class `Correct` and `ok` is a variable of this class. The expression

```
ok and v[i] > 0
```

would be equivalent to

```
ok.and( v[i] > 0 )
```

and should return `false` if `ok` if `false`, without evaluating "`v[i] > 0`". But by the semantics of method call the real parameter is always evaluated.

For short, Green does not support operator overloading because of the semantic differences between basic classes and reference classes and because of the impossibility of simulating some operators.

3.8 Expanded Variables

An instance or local variable may be declared as

```
var clock : @Clock;
```

to mean it obeys value instead of reference semantics, like the basic classes. The declaration alone allocates memory for the object, which need not to be dynamically allocated.

Variables declared with `@` are constants — it is a compiler error to use them in the left-hand side of an assignment. These `@`-variables correspond to variables of expanded classes in Eiffel or to non-pointer variables in C++. They are called *expanded* variables in Green.

An expanded variable or `@`-variable should be initiated by sending to it an `init`[2] message:

```
clock.init(13, 52);
```

This code replaces a call to method `new` of class `Clock` that would be made if `clock` were a regular variable:

```
clock = Clock.new(13, 52);
```

Messages `init` can only be sent to expanded variables. That will never cause an error because an expanded variable will always refer to an object of its declared class — an expanded variable cannot receive objects in assignments. Then, in particular, they cannot receive objects of subclasses which do not have a particular `init` method.

If `clock` were a regular variable and "`clock.init(13, 52)`" were legal, a runtime type error could occur. Variable `clock` could refer to an object whose class does not define an `init` method. These methods never belong, by definition, to a class type, which is the set of public method signatures.

Expanded variables always refer to the same object at runtime. This allows the compiler to optimize message sends to them. No runtime search for a method is necessary — a method call is made as efficient as a regular procedure call in procedural languages.

There should not be any cycles in the classes of expanded variables. That is, it is illegal a class `A` declare an expanded variable of class `B` and class `B` declare an expanded variable of class `A`.

[2]See Section 4.1.

Capítulo 4

Class Objects

4.1 Classes as Objects

In Green classes are also objects. They can be passed as parameters, stored in variables, and used in any place another object can. When we declare

```
class A
  ...
end
```

we are specifying the methods and instance variables that objects of class `A` will have. To declare the object that represents class `A` we write

```
object A
    // constructor
  public:
    // methods
  private:
    // methods and instance variables
end
```

`A` is the name of a classless object created before the program starts its execution. The identifier `A` is a read-only variable that refers to the class object. Then `A` is in fact a constant and can be used as labels of case statements.

Class objects play the role of metaclasses found in other object-oriented languages. To keep the language as simple as possible, we have chosen not to allow a class to have a class. This causes the infinite regression problem: a class should have a class that should have a class and so on.

Objects are allocated by methods called `new` of class objects. These methods are automatically created by the compiler. For each method `init` in class `A`[1] the compiler creates a `new` method in class object `A` with the same number of parameters and same types.

Each method `new` allocates memory for a class-`A` object, initializes some variables,[2] and sends to this object the corresponding message `init`. Then the code

```
f = Circle.new(30, 50);
```

will call method `Circle::init(integer, integer)` after allocating memory to the object and doing some internal work.

All methods called `init` are the constructors of the class. Like other methods they may be overloaded providing alternative ways for initializing objects of the class. They cannot have return

[1] There may be several because of overloading.

[2] Such as the hidden instance variable that refers to an array with pointers to the methods of the object class.

value types and cannot be abstract (see Section 3.6). `init` methods are always put before the `public` section of the class. They have special scope rules that do not match those of public or private sections.

If a class does not define any `init` method, the compiler will issue an error message since no object of this class can be created. If one wants this, she should declare the class as abstract.

A method called `new` may be declared in the public section of a *class object*. However, there should be no `init` method in the *class* with number of parameters and parameter types equal to the `new` method. If this were allowed, there would be a collision between the programmer's method `new` and the method `new` created by the compiler based on the `init` method of the class.

It is interesting noting a method `new` may be defined in a class. It will not differ from any other method in any way and will not be related to the constructors or to `new` methods of the class object.

The private part of a class object can be used inside the corresponding class:

```
object Person
  private:
    var minSalary : real = 800.0;
end

class Person
    proc init( pname : String; psalary : real )
      begin
      if psalary < Person.minSalary
      then
        salary = Person.minSalary;
      else
        salary = psalary;
      endif
      name = pname;
      end
  public:

    ...
  private:
    var name : String;
        salary : real;
    ...
end
```

This will never cause any error because `Person` is a read-only variable — it will always point to an object that has a `minSalary` variable. This variable is initialized in its declaration. This is allowed in class objects but not in classes. A class object defines an object and as such there is memory associated to it. Therefore variable `minSalary` exists since the start of the program. Instance variables in classes cannot be initialized in the declaration because classes are just types — there is no memory associated to them. Besides that, a type is a static declaration that should not be mixed with run-time commands as assignments.

Green supports a shortcut to the common assignment

```
b = B.new();
```

in which the type of `b` is `B`. This assignment can be written as

```
b#init();
```

This makes life easier in case of arrays (seen ahead) and long class names.

Although class object `Person` is related to class `Person`, they represent different things. Therefore, to use variable `minSalary` of class object `Person` inside class `Person` it is necessary to use the dot:

```
Person.minSalary
```

In fact, there are only two special relationships between a class object `A` and class `A`:

1. the compiler adds a method `new` to the class object for each method `init` of class `A` and;

2. class `A` can manipulate the private part of the class object — the opposite is not true.

Since a class object is created only once before the program starts, there is only one instance of it at run time. A class object `A` cannot be abstract even if class `A` is. A class object is an object and therefore represents a real entity which is, of course, not abstract.

A class object may declare a method `init` without parameters before its public section. This method is called after the creation of the class object before the program starts. There is no calling order among the `init` methods of the classes of a program. The `init` method of a class object can be called by any class object method, although this will be rarely necessary. Since the `init` method of a class object is not public, it cannot be called outside the class object itself.

4.2 Constant Declaration

Constants can be declared in the `public` or `private` section of a class object as in the example:

```
object Compass
  public:
    const
      North = 1,
      South = 2,
      East  = 3,
      West  = 4;
end
```

These constants are accessed using the dot, as in "`Compass.North`".

The type of a constant may be specified:

```
const
  MaxCh : integer = 64,
  Last  : char = '\0';
```

Otherwise, the constant type will be the same as the value type. Only constants of strings (Chapter 9) and basic types can be declared. The value after "`=`" may be an expression but the compiler should be able to evaluate it.

Constants cannot be declared in classes. If this were allowed we would have two different ways of declaring constants (in classes and class objects), which is bad. Besides that, a type declaration like a class should specify only the properties of each of its objects, not constants shared by all objects of a class. This is the job of class objects that are equivalent to metaclasses of other languages.

4.3 Enumerated Constants

Enumerate constants are declared as

```
enum(red, yellow, green, blue);
```

in a class object. This declaration is exactly the same as

```
const
  red = 0, yellow = 1, green = 2, blue = 3;
```

or

```
const
  red    : integer = 0,
  yellow : integer = 1,
  green  : integer = 2,
  blue   : integer = 3;
```

An integer value can be assigned to an `enum` constant:

```
enum(red, yellow = 5, green, blue);
```

This is equivalent to

```
enum(red, yellow = 5, green = 6, blue = 7);
```

4.4 Where the Program Execution Starts

The execution of a program starts in a method `run` of a class object specified at link time. Method `run` may take a `String` array as parameter:

```
proc run( args : array(String)[] )
```

This array is initialized with the program arguments before the program starts. Method `run` may not take parameters. Anyway, only one method `run` must be defined in the class object and in `one` of the signatures just described.

Capítulo 5

Inheritance

5.1 Introduction

Inheritance of a class `A` by a class `B` is set by the keyword `subclassOf`:

```
class B subclassOf A ... end
```

The inheritance of a class by another will be represented graphically as shown in Figure 5.1. It will be used an arrow from the subclass to the superclass. So, a superclass will be always above its subclasses in a figure. An object will be represented as a small circle and a reference of a variable to an object is represented as an arrow. See Figure 5.1 where variable `v` refer to object $\mathcal{Q}$.

Multiple inheritance is not allowed although multiple subtyping, defined ahead, is. In this report, we will call "superclasses of `A`" all direct and indirect superclasses of `A`.

A class may be specified as "final" using a compiler option.[1] Final classes cannot be subclassed and they do not have subtypes (defined elsewhere). That means a variable of a final class `Store` can only refer to `Store` objects, making optimizations much easier — there is no polymorphism in this specific case.

Some of the Green built-in or standard classes are final: `String` and all wrapper classes (`Char`, `Boolean`, `Byte`, `Integer`, `Real`, `Double`).

We have chosen not to introduce a `final` keyword in Green. This final mechanism is much like a compiler optimization and therefore belongs to the realm of language implementation. Object-oriented programming has no concept of "final" classes and real-world entities need not to be modeled as such.[2] Therefore, we decided to have final classes but in the domain of the compiler and not in the language. There may be even other optimizations like this and I hope to discuss how they could be made in a yet-to-be-made chapter about compile-time metaobjects.

[1] Yes, this is similar to Java final classes.

[2] Of course, one can always invent some class that must not have any subclass but that will be very rare.

Figura 5.1: a) Class `B` inherits from class `A`   b) Variable `v` refer to object $\mathcal{Q}$

5.2 The `Any` and `AnyClass` Classes

Class `AnyClass` inherits from `Any` and is automatically inherited by any class that does not explicitly inherit from another class. So, `Any` and `AnyClass` are superclasses of all other classes in any program, including the array classes. No object of these classes can be created — they are abstract. Class `Any` is the equivalent of `Object` in Smalltalk and Java.

Class `AnyClassObject` is an abstract class that inherits from `Any`. A variable of `AnyClassObject` can refer to any class object. For example, one can write

```
var a : AnyClassObject;
a = Circle;
```

in which `Circle` is a class. In this example, `AnyClassObject` may be replaced by `Any` although this is not generally `true` as will be seen elsewhere.

Class `Any` cannot be directly inherited since it lacks some methods all classes should inherit and that are defined in class `AnyClass`. Class `Any` defines the methods specified below. These methods will work correctly with any object, even though they do not know at compile time the structure of the object (its instance variables).

1. `toString() : String`

   returns the empty string . This should be redefined in subclasses to return a string representation of the object.

2. `isObjectOf( aClass : AnyClassObject ) : boolean`

   returns `true` if `aClass` is the receiver class or its superclass. If `Circle` is subclass of `Figure`, the `if` expression will always be true:

   ```
   var f : Figure;
   f = Circle.new(10, 30, 7);
   if f.isObjectOf(Figure) and
      f.isObjectOf(Circle) and
      not f.isObjectOf(Square)
   then
     ...
   endif
   ```

3. ```
   shallowClone() : Any
               ( exception : CatchOutOfMemoryException )
   ```

   creates and returns a new object of the same class as the object that received this message. The instance variables of the receiver are copied to the new object. If there is not sufficient memory for the operation, exception `OutOfMemoryException` is thrown.

4. ```
   deepClone() : Any
            ( exception : CatchOutOfMemoryException )
   ```

   creates and returns a new object of the same class as the receiver of this message. It creates a copy of each object referred directly or indirectly by the receiver and assembly the new object references to match that of the receiver object. If there is not sufficient memory for the operation, exception `OutOfMemoryException` is thrown.

5. `shallowCopy( other : Any ) : boolean;`

   copies the instance variables of the object pointed by `other` into `self`. If the classes of `self` and `other` are different from each other, this method returns `false`.[3] No object is created by this method.

6. `shallowEqual( other : Any ) : boolean`

   compares the instance variables of `self` and `other` using the operator `==` and returns `false` if some test returns `false`.

   The operator `==`, when applied to two variables whose types are classes, returns `true` if the two variables point to the same object.

7. `deepEqual( other : Any ) : boolean`

   makes a deep equality test between `self` and `other`. This method returns `false` if the object layout of `self` is different from `other` or if there is an instance variable whose type is a basic type in an object referred directly or indirectly by `self` whose value is different from the value of the corresponding instance variable in the objects pointed directly or indirectly by `other`.

8. `getInfo() : AnyObjectInfo`

   returns an object that describes `self`. This method is explained in Chapter 16 and Appendix B.

9. `equals( other : Any ) : boolean`

   returns `true` if `self` is equal to `other`. The body of this method is

   ```
   return self == other;
   ```

   which results in an address comparison. Each class should redefine this method to give it a better meaning.

Note `deepCopy` is missing from this listing. We believe it would be rarely used.
Class `AnyClass` defines the following methods:

- `getClassInfo() : ClassInfo`

  returns an object that describes the class of the object.

- `getClassObject() : AnyClassObject`

  returns the class of the receiver which is a classless object. For example, the `if` expression that follows will always be true.

  ```
  var c : Circle;
  c = Circle.new(10, 20, 10);
  if c.getClassObject() == Circle then ... endif
  ```

Chapter 16 and Appendix B discusses these methods in depth.

[3]Note that `shallowCopy` returns `false` even when the class of object `other` is a subclass of the class of `self`.

5.3 Method Look-up and `super`

The statement

`a.m(`p_1`, `p_2`, ... `p_n`)`

is the sending of the message `m(`p_1`, `p_2`, ... `p_n`)` to the object `a` refer to. The message name is `m` and p_1, p_2, ... p_n are its arguments (parameters).

At run time, this message send orders the run-time system to look for a method called `m` in the class of the object `a` refer to. If no method with this name is found, the search continues in the superclass of the object class and so on. As we will see in Chapter 7, the compiler guarantees that a message will be found at run time if no compiler error has occurred. This run-time search for a method is called dynamic method look-up.

The statement

`super.m(`p_1`, `p_2`, ... `p_n`)`

in a class `B` makes the compiler look for a method `m` beginning in the superclass of `B`, that we will call `A`. If `A` does not define a method `m`, the search continues in the superclass of `A` and so on. This search is made at compile time and results in a direct call to a method `m` of a direct or indirect superclass of `B`. The receiver of this message is `self`. Of course, there will be an error if the method found is abstract.

An `init` method of a superclass can be called using `super`, since a message to `super` is a message to `self` with a fixed place to begin the search for the method.

5.4 The `subclass` Section

Subclasses cannot access private instance variables of the superclass. If it is necessary to reveal private information of a class to its subclasses, the programmer should declare methods to access this information in the `subclass` section of the class. See Figure 5.2. Note class `Circle` accesses method `setX` through a message to `self`. If `Circle` had a method

```
proc medX( other : Circle ) : integer
  begin
  return (x + other. getX())/2;
  end
```

there would be a compiler error in "`other.getX()`". The reason, as will be seen in Chapter 7, is that `other` may be referring to an object that is not subclass of `Circle` and that may not have a method `getX`.

Dynamic look-up is also made with methods of the subclass section. To show that, suppose class `Circle` defines a method `setX` in its subclass section. Then the code

```
var c : Circle = Circle.new(10, 50);
c.walkTo(20, 40);
```

calls method `Figure::walkTo` (since `walkTo` is not redefined in `Circle`) which sends message "`setX(newX)`" to `self`. The executed method will be `Circle::setX`.

There should be at most one subclass section in a class declaration. It should be after the public and before the private section. When creating a subclass `B` of a class `A`, some restrictions apply in the redefinition or definition of methods:

1. a public method of `A` may be redefined only as a public method of `B`;

```
class Figure
  public:
    proc walkTo( newX, newY : integer )
      begin
      setX(newX);
      ...
      end
    ...

  subclass:
    proc setX( newX : integer )
      begin
      x = newX;
      end
    proc getX() : integer
      begin
      return x;
      end
    ...

  private:
    var x, y : integer;
end

class Circle subclassOf Figure
  public:
    proc move( newX, newY : integer )
      begin
      setX(newX);  // ok, calls Figure::setX
      ...
      end
    ...
end
```

Figura 5.2: Use of `subclass` section

2. a subclass method of `A` may be redefined only as a subclass method of `B`;

3. a private method of `A` may be redefined in any section (public, subclass, or private) of `B` since no one outside `A` should or could know of the existence of its private methods. Of course, the `A` and the `B` method would be completely unrelated to each other. As an example, suppose class `A` defines a public method `m` and a private method `p`. Inside `m` there is a message send "`self.p()`". Class `B` inherits from `A` and defines a public method `p`. The code
```
var b : B;
b = B.new();
b.m();
```
will call method `A::m` which sends message "`p()`" to `self` that calls method `A::p`. If `p` were public, the method called would be `B::p`.

Restriction 2 could be relaxed to allow a subclass method of `A` to be redefined as a public method of `B` but that would not add any power to the language and would make it harder to understand (so we believe).

The subclass section may declare abstract methods which should be redefined in subclasses. It does make sense to have abstract methods in the subclass section because dynamic look-up is also made with methods of this section.

5.5 `nil`

`nil` is a special global variable that points to an object of the predefined class `Nil`. It can be assigned to any variable whose type is a class. `Nil` is considered to be a subtype of any other type, which means `nil` can be assigned to any variable whose type is a class. Albeit that, class `Nil` has no method and `Nil.new()` always returns `nil` — see class `Nil` below.

```
object Nil
  public:
    proc new() : Nil
      begin
      return nil;
      end
end

class Nil
end
```

It is a run-time error to send a message to any object of class `Nil`. If this happens an exception `MessageSendToNilException` will be thrown.

5.6 Abstract Classes and Inheritance

Two points are of interest here:

- a subclass of an abstract class can call the superclass constructors using `super`;

- if a class inherits from an abstract class and does not define all method bodies of inherited abstract methods, then this class should also be declared abstract.

5.7 Abstract Classes, Assertions, and Inheritance

Assertions may be specified even to abstract methods. Class `Container` defined in Figure 3.4 is redefined below with some assertions. If a class inherits from `Container` and does not define new assertions for an inherited method, this method will use the inherited assertions.

```
abstract class Container
  public:
    abstract proc add( x : integer )
      assert
        before not full();
      end

    abstract proc get() : integer
      assert
        before not full();
      end

    proc empty() : boolean
      begin
      return getSize() == 0;
      end

    proc full() : boolean
      begin
      return getSize() == getMaxSize();
      end

    abstract proc getSize() : integer
      assert
        after  result >= 0 and result <= getMaxSize();
      end

    abstract proc getMaxSize() : integer;
end
```

Suppose class `List` inherits `Container`:

```
class List subclassOf Container
  public:
    proc add( x : integer )
      begin
      n = n + 1;
      v[n] = x;
      end
    ...
end
```

Since method `put` does not define an assertion clause, it inherits the assertion clause of `Container` method `put`.

As noted by Meyer [10], the use of assertions with abstract classes allows one to describe the program design and its specification at the same time.

Capítulo 6

Arrays

6.1 Introduction

An array `ac` of `char` is declared as

```
var ac : array(char)[];
```

The number of array elements should not be specified. Arrays are objects in Green and as such they are dynamically allocated with `new`:

```
ac = array(char)[].new(10);
```

Note that the number of elements should be specified as a parameter to `new`. An array of any basic type has a `fill` method to fill all array positions with a given value:

```
v.fill(0)
```

To access the ith array element, one should use `v[i]`. Indices range from `0` to `n - 1` where `n` is the number of array elements.

A two-dimensional integer array is also declared without specifying the number of lines/columns:

```
var mi : array(integer)[][];
```

The dimensions are given in the allocation:

```
mi = array(integer)[][].new(10, 30);
```

It is also possible to give only the first array dimension:

```
mi = array(integer)[][].new(10);
```

This allocates ten empty lines that can be separately created:

```
for i = 0 to mi.getSize() - 1 do
    mi[i] = array(integer)[].new(30);
```

`getSize` returns the number of array elements.

Although the above code creates ten arrays with the same size (30), it could have created arrays of different sizes. In fact, the number of elements of each dimension does not belong to the array type. As a consequence, a method

```
proc mult( mr : array(real)[][] )
```

accepts as parameter any two-dimensional `real` array.

The operator `[]` is used to access array elements. Then,

`mr[0]` is an array whose type is `array(real)[]`

`mr[i][j]` is a real number.

Since the number of elements of each dimension does not belong to the array type,

- the size of each dimension is never specified in a variable or parameter declaration;
- the dimensions are just instance variables of the array class and therefore should be initialized by the `new` method.

Arrays are a kind of parametrized classes specially tuned for efficiency. By the discussion above, an array depends only on its element type and the number of dimensions and these would be parameters to a non-existing generic class `Array`.

6.2 Methods of Array Classes

An array of a basic class, like `array(char)[]`, inherits from abstract class `AnyArray`. An array of a non-basic class, like `array(Person)[]`, inherits from abstract class `AnyClassArray`, which inherits from `AnyArray`. Class `AnyArray` defines the following methods:

```
getSize() : integer
```
returns the array size.

```
set( v : Any; i : integer )
   ( exception : CatchAnyArrayException )
```
set to `v` the `i`th position of the array. The `exception` variable is used to throw exceptions, described in Chapter 11. The methods that follow work similarly with arrays of two or more dimensions. These methods may throw the following exceptions:

- `TypeErrorException`. This exception is thrown if there is a type error; that is, if the array element type is `T` and the run-time type of object `v` is not a subtype of `T`;
- `TooManyDimensionsException`, which is thrown if the array has less dimensions than those specified. For example, if one uses the next method `set` with a one-dimensional array. There should have been specified only only index, `i` or `j`.

```
set( v : Any; i, j : integer )
   ( exception : CatchAnyArrayException )
```

```
set( v : Any; i, j, k : integer; others : ... array(integer)[] )
   ( exception : CatchAnyArrayException )
```

```
get( i : integer ) : Any
   ( exception : CatchAnyArrayException )
```
returns the element of the array index `i`. The methods that follow work similarly with arrays of two or more dimensions. The `get` methods may throw exception `TooManyDimensionsException`.

```
get( i, j : integer ) : Any
   ( exception : CatchAnyArrayException )
```

```
get( i, j, k : integer, others : ... array(integer)[] )
   ( exception : CatchAnyArrayException )
```
See Section 6.5 for the meaning of "`...`".

```
toString() : String
```

returns a string like "`array(T)[][]`" if the array has two dimensions and `T` as the element type.

Each array of a basic class defines the following methods and constructors. `T` is the array element type. Then, in `array(char)[]`, `T` represents `char`.

- `init( first, second, third, ... : integer )`

  Constructor of an array in which `first`, `second`, ... are the array dimensions. An array of `n` dimensions has `init` methods from `1` to `n` parameters. As in the example
  ```
  mi = array(integer)[][].new(10);
  ```
  in which method
  ```
  proc init( first : integer )
  ```
  is called by `new`, the minor dimensions need not to be specified in the array creation. Later on, the program must allocate memory for these minor dimensions in order to use the array;

- `getIter() : DS.Iter(T)` returns an iterator for the array. Iterators are presented in Chapter 15. `T` is the type of the array elements. "`DS.Iter(T)`" is the type of the iterator;

- `forEach( f : Function(T) )`

  calls "`f.exec`" on each array element. Class `Function` is defined in Chapter 15. `T` is the type of the array elements;

- `replaceBy( cmd : Command(T) )`

  replaces each array element `x` by "`cmd.doIt(x)`". Class `Command` is defined in Chapter 15. `T` is the type of the array elements;

- `collect( f : Filter(T) ) : array(T)[]`

  collects all array elements `x` such that "`f.test(x)`" evaluates to `true`. These elements are inserted in a new array returned by this method. `T` is the type of the array elements. `Filter` is defined in Chapter 15;

- `remove( f : Filter(T) )`

  removes all array elements `x` such that "`f.test(x)`" evaluates to `true`;

- `reset( up : boolean )`

  initiates the array iterator. If `up` is true, method `next` will yield array elements from index `0` to `getSize() - 1`. If `up` is `false`, the elements will be yield from `getSize() - 1` to `0`;

- `reset()`

  the same as `reset(true)`;

- `more() : boolean`

  returns `true` if there is more elements to be yield by the iterator;

- `next() : T`

  returns the next array element according to the order set by calling `reset`. The counter of the iterator is incremented. An example of use of an iterator is:

```
var vi : array(integer)[];
     k : integer;
...
vi.reset();
while vi.more() do
  Out.writeln( vi.next() );
```

- `fill(value : T)`

  fill all array positions with value.

Class `AnyClassArray` defines the following methods:

- `getIter() : DS.Iter(Any)`
- `forEach( f : Function(Any) )`
  `(exception : TypeErrorException)`
- `replaceBy( cmd : Command(Any) ) (exception : TypeErrorException)`
- `collect( f : Filter(Any) )`
  `(exception : TypeErrorException) : array(Any)[]`
- `remove( f : Filter(T) )`

  removes all array elements `x` such that "`f.test(x)`" evaluates to `true`;
- `reset( up : boolean )`
- `reset()`
- `more() : boolean`
- `next() : Any`

Some methods will throw exception `TypeErrorException` if a type error occurs. Each of these methods play a role equal to the method of same name added to every basic-class array.

The compiler adds a `init` method, as defined to basic-class arrays, to all arrays.

6.3 Initialization and Creation of Arrays

An array can be initialized in its declaration through a special syntax:

```
proc selectFriend( day : integer ) : String
  begin
  var friends : array(String)[] = #( "Tom", "Jo", "Anna", "Peter" );

  return friends[ day%friends.getSize() ];
  end
```

Note that `selectFriend` *could not* have been written as

```
proc selectFriend( day : integer ) : String
  var friends : array(String)[] = #( "Tom", "Jo", "Anna", "Peter" );
  begin
  return friends[ day%friends.getSize() ];
  end
```

since assignments are not allowed in the local variable declaration section before `begin`.

Arrays can also be initiated with constants in the private section of a class object:

```
object Month
  public:
    proc get( i : integer ) : String
      begin
      return strMonth[i];
      end
  private:
    var strMonth : array(String)[] = #( "Jan", "Feb", "Mar", "Apr",
         "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dez" );
end
```

An array object, like any other, can be created using `#` :

```
var v : array(integer)[];
v#init(10);
```

This is the same as

```
v = array(integer)[].new(10);
```

6.4 Arrays and Inheritance

As already seen, class `AnyArray` defines methods for getting and setting array elements, as shown in the following example.

```
var aa : AnyArray;
var vi : array(integer)[];
var mr : array(real)[][];

vi#init(40);
mr#init(30, 50);
vi[0] = 1;
mr[0][0] = 7.0;

aa = vi;
var i : integer = integer.cast(aa.get(0));

aa = mr;
var r : real = real.cast(aa.get(0, 0));

aa.set(3.0, 0, 0);          // aa[0][0] = 3.0
Out.writeln( mr[0][0] );    // 3.0
```

Methods `get` and `set` throw an exception if their parameters are not type correct.

Although array classes are much like other classes, they cannot be subclassed. If they could, arrays could not be efficiently compiled.

Every array class inherits from `AnyArray` and is subtype only of `AnyArray`. Therefore an array "`array(Figure)[]`" is not a supertype or superclass of "`array(Circle)[]`", even `Circle` being a subclass of `Figure`. If it were, as in Java, there could be a run-time type error — see the code below.

```
var vf : array(Figure)[];
var vc : array(Circle)[];

vc#init(10);
  // suppose this is allowed
vf = vc;
vf[0] = Square.new(10);
  // now vc[0] refer to a square
Out.writeln( vc[0].getRadius() );   //  message not found
```

6.5 Methods with Variable Number of Parameters

A method that takes an undetermined number of parameters may be specified as:

```
proc print( whiteSpace : integer; v : ... array(Any)[] )
```

In this example `print` has one fixed parameter `whiteSpace` and any number of parameters following it, represented by array `v`. This array should always be at the end of the parameter list. Each of the variable number of real parameters should be a subtype of `Any`, which includes any object. Basic type objects as integers and characters are packed in wrapper objects of classes `Integer`, `Char`, etc. These classes are defined in Chapter 8. In a call to `print`, the real parameters after the first one are used to initialize an array passed as a parameter to this method.

A complete example of definition and use of method `print` is given next.

```
object MyScreen
  public:
    proc print( whiteSpace : integer;
                v : ... array(Any)[] )
      assert
        before whiteSpace >= 0 and whiteSpace < 80 and v.getSize() > 0;
      end

        var i : integer;
      begin
      for i = 0 to v.getSize() - 1 do
        begin
        Out.write( getWhiteSpace(whiteSpace) );
        Out.writeln( v[i].toString() );
        end
      end
   ...
end
```

This example uses object `Out` and its method `writeln` to do the output to the screen. Class `MyScreen` can be used as

```
MyScreen.print( 5, "i = ", i, "(", 2, ")" );
```

in which `whiteSpace` would receive `5` and the other parameters would be packed in an array. Parameter `i` (which belongs to type `integer`) and number `2` would be used to create `Integer` objects that would then be inserted in the array. Classes like `Integer` are called *wrapper classes* and are discussed in Section 8.2.

It may be interesting to restrict the parameter of the undefined part to subtypes of a given class as is made in the next example.

```
abstract class Drawable
  public:
    abstract proc draw()
end

object MyScreen
  public:
    proc drawAll( v : ...array(Drawable)[] )
        var i : integer;
      begin
      for i = 0 to v.getSize() - 1 do
        v[i].draw();
      end
    ...
end
```

Note that a call

```
Screen.drawAll( circle, 5 );
```

would result in a compile time error since class `integer` (`5`) is not subtype of `Drawable`.

There may not be in the same class two methods like

```
proc m( i : integer; ch : char; v : ... array(Any)[] )
proc m( j : integer; ch : char )
```

They differ only in the last parameter `v` that indicates a variable number of parameter. If this were allowed there would be an ambiguity in the call

```
x.m(5, #'A');
```

Either method `m` could be used.

6.6 Expanded Arrays

Array variables can be declared with `@` if the array dimensions are specified:

```
var clockArray : @array(Clock)[100];
var matrix     : @array(integer)[MaxLin][MaxCol];
```

`MaxLin` and `MaxCol` must be constants. We expect expanded array variables will be very rarely used. They are supported to make the language orthogonal.

In the declaration of `clockArray`, `Clock` is not preceed by `@`. That means the `Clock` objects should be allocated with `new`:

```
clockArray[0] = Clock.new(13, 52);
```

One can choose to allocate all `Clock` objects by using `@Clock` as the element type:

```
var clockArray : @array(@Clock)[100];
```

```
clockArray[0].init(13, 52);
clockArray[0] = Clock.new(0, 0); // error !!
```

The last line is a compile-time error since @-variables cannot receive objects in assignments.

One may also declare

```
var clockArray : array(@Clock)[];
```

Here `clockArray` is a dynamically-allocated array of constant objects. This should be preferable to the declaration

```
var anotherClockArray : @array(@Clock)[100];
```

`clockArray` should be created with `new`:

```
clockArray = array(@Clock)[].new(max);
clockArray[0].init(13, 52);
```

Capítulo 7

The Green Type System

7.1 Types and Subtypes

The *type* of a class is the set of signatures of its public methods. The signature of a method is its name, return value type (if it returns a value), and formal parameter types (parameter names are discarded). For example, the type of the class `Store` of Figure 3.3 is given by

`typeOf(Store)` = { `set(integer), get() : integer` }

Throughout this report `typeOf(A)` will be the type of class `A`.

A type $S = \{\mathtt{n}_1,\ \mathtt{n}_2,\ \ldots \mathtt{n}_p\}$ is equal to type $T = \{\mathtt{m}_1,\ \mathtt{m}_2,\ \ldots \mathtt{m}_q\}$ ($S = T$) if $p = q$ and $\mathtt{n}_i = \mathtt{m}_i$, $1 \leq i \leq p$. The relation = for methods is defined as follows.

Let

$\mathtt{n}(T'_1,\ T'_2,\ \ldots T'_k)\ :\ U'$
$\mathtt{m}(T_1,\ T_2,\ \ldots T_t)\ :\ U$

be the signatures of two methods. U and U' are return value types. We say that $\mathtt{n} = \mathtt{m}$ if

- `m` and `n` have the same name.
- $t = k$.
- $T_i = T'_i$, $1 \leq i \leq t$.
- $U' = U$.

If necessary, the method signatures of S and T should be arranged in an order such that the relation above becomes true. That is, the order the signatures appear in the set does not matter, although the order of parameter/return value types of a signature does matter.

The type equality is a recursive definition because it uses the method definition and vice-versa. An algorithm to test if two types are equal could never finish its execution because of an endless recursion. However, that never happens according to the Proposition below.

Proposition 7.1.1 *An algorithm that tests if two types are equal according to the language definition will always finish its execution.*

Proof:

The algorithm `isEqual` of Figure 7.1 compares two types and returns `true` if they are equal. Algorithm `isEqual` calls `isEqualTo` that performs a Depth-First Search in both types at the same time, comparing the corresponding vertices. Therefore, it will finish its execution. ❐

```
proc isEqual( S, T : Type )
  var Ig : Set of Tuples (X, Y) where X and Y are types;

begin
Ig = empty set;
return isEqualTo( S, T, Ig );
end

proc isEqualTo( S, T : Type; Ig : Set )
begin

  // (S,T) is inserted into Ig when S = T
if (S, T) belongs to set Ig
then
  return true;
else
  if S or T is a basic type
  then
    return S = T;
  endif
endif
insert (S, T) into Ig;
for each method m(U_1, U_2, ...  U_{k-1}) :  U_k of S, do
  if T does not have method with the same name, number of parameters
    and return values as method m of S
  then
    return false;
  else
    assume assume V_i is the type of the method m of T corresponding to U_i of S
    for i = 1 to k do
      if not isEqualTo( V_i, U_i, Ig)
      then
        return false;
      endif
  endif
return true;
end
```

Figura 7.1: Algorithm to discover if two types are equal

A type $S = \{\texttt{n}_1,\ \texttt{n}_2,\ \ldots\texttt{n}_p\}$ is a subtype of type $T = \{\texttt{m}_1,\ \texttt{m}_2,\ \ldots\texttt{m}_q\}$ (we will use $S \prec T$ for that) if $p \geq q$ and $\texttt{n}_i = \texttt{m}_i$ for $1 \leq i \leq q$. By this definition, $S \prec T$ implies that $T \subset S$. That is, a subtype has at least the same method signatures as its supertype. Since $X \subset X$ for any type X, any type is also subtype of itself. We usually say "class `B` is a subtype of class `A`" instead of "the type of class `B` is a subtype of the type of class `A`".

When class `B` inherits from class `A`, `B` is a subclass of `A`. So, `B` inherits all public methods of `A`, implying that `B` is a subtype of `A`. Class `B` can redefine an inherited method from `A` but its signature should be the same as in the superclass. By this type definition, any subclass is also a subtype, but it is possible to have a subtype that is not a subclass.

This type system is a restriction of that of POOL-I [1] language. The programs that are type correct according to the type system defined above are also type correct according to the POOL-I type system since this has less restrictions than that. POOL-I follows the Cardelli [2] rules for subtyping. To explain Cardelli's rules, suppose class `B` is subtype of class `A` that defines a method

```
proc m( x : C ) : D
```

Class `B` can define a method

```
proc m( x : C′ ) : D′
```

such that $\texttt{C} \prec \texttt{C}'$ and $\texttt{D}' \prec \texttt{D}$. In our type system, `C` must be equal to `C′` and `D` must be equal to `D′`.

Some object-oriented languages such as C++ [17] and Eiffel [10] [11] associate subtype with subclass. A subtype of a class can be created only by subclassing the class. This results in a less flexible type system than that of Green. As an example, suppose there is a method

```
proc m( x : A )
```

of some class (it does not matter). In Eiffel and C++, method `m` accepts only objects of class `A` and its subclasses as parameters. In the type system of Green, method `m` accepts objects of any class that is subtype of `A`. Since there are potentially more subtypes than subclasses of `A`, the Green type system supports a higher degree of polymorphism than the Eiffel/C++ type system. The separation between the subtype and subclass hierarchy gives to a statically typed language much of the flexibility of untyped languages. This kind of type system is employed in the languages POOL-I [1], Sather [16] [6], Java [8],[1] and School [13]. The language Emerald [12] also uses this subtype relationship although it does not support inheritance.

Snyder [14] [15] asserts that there is a violation of encapsulation if subtyping is tied to subclassing. To show that, suppose class `B` inherits from `A` and a class-`B` object is used as a parameter to method `m` whose signature was shown in the previous paragraph. If class `B` is modified to not inherit from `A`, although keeping its interface, the code that passes class-`B` objects to `m` becomes type incorrect. This means that the inheritance of `A` by `B` is not private to `B`: it is public and cannot be changed without program modifications.

The type of a class can be discovered without compiling its file if there is no syntax error. This is possible since the type of a class is just its interface. It does not depend on the semantic correctness of the code inside the class methods. Then we have the following assumption:

Assumption 7.1.1 *It is possible to find the type of a class by doing a syntactic analysis in it if there is no syntax error in the file the class is.*

7.2 Rules for Type Checking

The statement

```
aa = bb
```

where the type of `aa` is a class, makes the variables `aa` and `bb` refer to the same object. The declared

[1] In fact, a restricted version of it.

type of `aa` and `bb` must be in the relationship

`type(bb)` $\prec$ `type(aa)`

where `type(x)` is the type of class `Z` considering `x` is declared as

```
var x : Z;
```

That is, assignments of the kind

```
Type = Subtype
```

are valid.

Assignments of the kind

```
Type = Supertype
```

are allowed if made through method `cast`:

```
b = B.cast(a);
```

This method is automatically added by the compiler to each class object (see Section 10.3) and it throws exception `TypeErrorException` if object `a` cannot be cast to class `B`. Method `cast` supplied by the compiler will succeed if the class of object `a` at run-time is a subtype of `B`.

The statement

`a.m(`p_1`,` p_2`, ...` p_n`)`

sends the message `m(`p_1`,` p_2`, ...` p_n`)` to the object `a` refer to and is legal if:

- method `m(`T_i`,` T_2`, ...` T_n`) :` U_{n+1} belongs to `type(a)`, where `type(a)` is the declared type of `a` and;
- for $1 \le i \le n$,
  - `type(`p_i`)` $\prec T_i$, where `type(`p_i`)` is the declared type of p_i and T_i is a class name or
  - p_i is a basic value (`char`, `integer`, ...) and can be automatically converted to T_i.[2]

The run-time system searchs for a method named `m` in the class of the object. If no method by this name is found, it searchs in the object superclass, then in the superclass of superclass and so on. If method `m` is not found, an error occurs. To send a message to an object that cannot respond to it is a *type error*.

The keyword `self` inside a method code evaluates to the object that received the message that caused the execution of the method. A message send to `self` is considered by the type system as a normal message send in which `type(self)` is the type of the class in which this statement is with the addition of all methods of the:

- `private` section;
- `subclass` sections of this class and all superclasses.

The message send

`super.m(`p_1`,` p_2`, ...` p_n`)`

in a method of a class `B` is considered type correct if

- the superclass of `B` is `A` and `m` does not belong to the private section of `A` and;
- the message send

  `self.m(`p_1`,` p_2`, ...` p_n`)`

  inside an `A` method would be correct.

[2]Currently no automatic cast between basic types is allowed.

The a variable in the statement a.m(p_1, p_2, ... p_n) may refer to an object whose type is a subtype of its declared type because of assignments of the form `Type = Subtype`. But that does not cause any error, since a subtype has all methods of a class with exactly the same method signatures.

The m method of class

```
class A
  public:
    proc m( x : Store )
      begin
      x.set(5);
      v = x.get();
      end
    ...
  private:
    var v, k : integer;
end
```

takes a formal parameter x of class Store. Using the above rules, the compiler enforces that:

- class Store has methods corresponding to the message sends to x[3] inside method m, that is, set and get.
- the types of the real parameters can be converted into the types of the formal parameters. Remember there is no automatic coercion among the basic types.

7.3 Discussion on Types

In parameter passing to methods/procedures, there is an implicit assignment

`formal parameter = real parameter`

since any parameter is passed by value. The same is valid for method return values. Therefore, any type analysis can be restricted to assignments since this embodies parameter passing and return values of methods.

When a variable is declared, the programmer must give a class name that represents its type, as in

`var a : A;`

Then variable a can refer to objects of any subtype of class A. In fact, A does not represent the class A but the type type(A). That is the reason we used "A" as the type of the variable instead of typeOf(A).

Since this variable can refer to objects of classes that are subtypes of typeOf(A), the compiler does not know the size of the object that the variable will refer to at run time. This is the reason the declaration of variable a does not automatically allocate memory for a class-A object as in other languages.

This is also the reason why, inside a class-A method, we cannot access an instance variable x of a parameter p that has type A, as in

```
proc m( p : A ) // method of A
  begin
```

[3] We say "message send to x" to mean "message send to the object x will refer to at run time" since a message is sent to an object, not to a variable.

```
    p.x = 10;
    end
```

`x` can refer at run time to an object that does not belong to class `A` or its subclasses and therefore does not have instance variable `x`.

7.4 Assertions and Subclassing

Assertions are inherited by subclasses. If a subclass
overrides a superclass method and does not define an `assert` clause for this method, the assert clause of the superclass method is added to the subclass method by the compiler.

The `before` and `after` expressions of an assert clause of a subclass method should be semantically related to the corresponding expressions of the superclass method. This can be better understood by studying the relationship among types in inherited subclass methods, described next.

It could be possible to redefine the parameter types of methods in subclasses. For example, a method

```
proc whoEats( food : Vegetable ) : Animal
```

of a class `AnimalWorld` could be redefined in a subclass `MammalWorld` as

```
proc whoEats( food : Food ) : Mammal
```

in which `Vegetable` is subtype of `Food` and `Mammal` is subtype of `Animal`. Green does not support this feature because

- it is not largely used. In most cases we want to use a subtype as the parameter type of a subclass method, like use `Tomato` instead of `Food` as the type of `food` in a subclass method. This is not allowed — we could use a supertype as the parameter type. When redefining the return value type in a subclass method, we can use a subtype of the return value type of the superclass method (as `Mammal` and `Animal`);

- it would conflict with overloading. In Green two methods with different parameter types are considered different.

To understand why the sub and superclass method types should be related as described previously, we will study the example below.

```
var a : AnimalWorld;
var b : MammalWorld;
a#init();
b#init();
var v : Vegetable = Vegetable.new("Grass");
var animal : Animal;

animal = a.whoEats(v);
a = b;
animal = a.whoEats(v);
```

The last statement is a message send to a `MammalWorld` object. Then it should take a `Food` object as parameter and there is an implicit assignment

```
food = v;
```

This is correct since `food` type (`Food`) is a supertype of `v` type (`Vegetable`). Method `whoEats` of `MammalWorld` returns a `Mammal` object that can be assigned to an `Animal` variable.

Returning to assertions, a subclass method should make weaker assumptions on its parameters and stronger assumptions on its returned value [10]. As an example, a class `Function` defines a mathematical function and has a method

```
proc findRoot( firstGuess : real ) : real
  assert
    before Math.abs( f(firstGuess) ) < 1;
    after  Math.abs( f(result) ) < 1E-5;
  end
```

to find a function root. `Math` is a class object that defines mathematical functions like `sin`, `cos`, and `abs` which returns the absolute value of a number. Method `f` of `Function` evaluates the function with the given parameter.

`firstGuess` parameter of `findRoot` is a first guess of the root and the function value at this point should be at a distance less than `1` from `0`. Variable `result` holds the value returned by the function and therefore `f(result)` should be very close to zero.

A subclass `Polynomial` of `Function` could define a method

```
proc findRoot( firstGuess : real ) : real
  assert
    before Math.abs( f(firstGuess) ) < 5;
    after  Math.abs( f(result) ) < 1E-20;
  end
```

that uses a much better algorithm to find the root. It requires a weaker assumption on the parameter and produces a better result (stronger assumption). One can match superclasses with weaker assumptions because a superclass is more abstract than its subclass. A superclass may has (and generally has) less methods and represents more concepts than its subclasses.

One can match subclasses with stronger assumptions by the same reasons. The result is that `before` and `after` expressions should obey similar requirements of parameter and return value type redefinition in subclass methods. As an example, the code

```
class Example
  public:
    proc calc( f : Function )
        var r : real;
      begin
      r = f.findRoot(1.0);
      Out.writeln(r);
      end
    ...
end  // Example

...
var e : Example;
var p : Function;

e#init();
p = Polynomial.new("x*x - 2*x + 1");
e.calc(p);
```

makes variable `p` refer to a polynomial object and calculates a root with a better precision than the creator of class `Example` expected. Since the programmer is using a `Function` parameter, she should expect the worst precision `1E-5` than what she really gets, `1E-20`. Similar reason applies to the parameter value: any value good for `Function::findRoot` is also good for `Polynomial::findRoot`.

7.5 Constructors, Inheritance, and Types

A constructor (`init` method) of a class `A` can be called by

- a compiler-created `new` method of class object `A`;
- sending a message to `self` inside class `A`;
- sending a message to `super` in a subclass of `A`;
- sending a message to an expanded variable.

Polymorphism *does work* with constructors. In any message send "`self.init(...)`" or "`super.init(...)`" the search for a `init` method is made at runtime. Then, a constructor of a superclass can call a subclass constructor. This is probably wrong because subclass constructors can (and should) expect the superclass instance variables have been initiated. Besides that, endless loops can occur since subclass constructors usually call superclass constructors unconditionally. In this case the superclass constructor calls the subclass constructor creating thus a cycle.

Unlike other languages as C++ or Java, Green does not require a subclass to call the superclass constructor. Albeit that, not to call the superclass constructor is a bad practice and we expect the compiler will issue a warning if this will never or may not occur at run time. The reason for not introducing this feature is that it would require a special syntax. Then one could call a method `init` by two different ways and that would be confusing. Anyway, in most of the cases a subclass `init` method will call the superclass `init` in its first statement.

Since the programmer cannot send a `init` message to any object but `self` (even when through `super`), in fact `init` does not belong to the type of the class. This makes sense because the parameters of the constructors can reveal a lot about the class implementation.

The implementation of a class (its instance variables and method bodies) is highly subject to changes, and therefore the class `init` methods are too. The `init` methods are closely related to instance variables — in general the `init` parameters are assigned directly to instance variables without any processing. If the `init` methods belonged to the class type, a change to the class implementation would invalidate working code since this depends mainly on the class types. As the `init` methods do not belong to the types, any change in them requires only modification in the code

- that creates (with `new`) objects of the class;
- of subclasses of the class that calls the superclass constructor.

As an example of the problem above, suppose there are two `Queue` classes, `SQueue` and `DQueue`, with equal types. If the implementation of `SQueue` were modified from a linked list to an array, there could be added a method

```
proc init( maxSize : integer )
```

to set the maximum number of elements in the Queue. Now `SQueue` would have a type different from `DQueue` if `init` belonged to the class type.

All `init` methods must be declared before the class public section. This prevents dirt tricks as to put some or all constructors in the `subclass` section. Then only subclasses could call the `init` methods. If all `init` methods were put in the subclass section, only subclasses could create objects of the class.[4] This is not allowed because:

- it would be redundant with the module system that already offers visibility control;
- it makes the code hard to understand since it adds to the constructs more responsibilities than they were supposed to have.

When a message is sent to `self` inside a class, the method called may belong to a subclass of this class. In a `init` method, a message send to `self` could call a method of a subclass and this most probably will result in an error because:

- the subclass instance variables may be accessed by this subclass method called by `init` and;
- the subclass instance variables may not have been initialized since in the normal order of initialization superclasses are initialized before subclasses. In most cases the first statement of a `init` method is a call to a superclass `init` method.

To avoid this kind of error we hope the compiler will issue a warning if there is a message send to `self` inside a method `init`. This error may occur even if the message send refer to a private method because this method may call a public or subclass method with which dynamic look-up is made. Passing `self` as parameter in a message send inside `init` is also dangerous because the methods called can send a message to the `self` passed as parameter. And then a subclass method can be called.

7.6 Types and Overloaded Methods

Suppose the type of variable `screen` is `MyScreen` that defines several methods called `print` — an overloaded method.

In a message send

```
screen.print(x)
```

the method to be called, among all overloaded `print` methods, is defined at compile time. The declared type of `x` must be equal to a parameter type of one of the `print` methods. As an example, suppose class of `screen` has the following `print` methods

```
proc print( f : Figure )
proc print( c : Circle )
proc print( i : integer )
```

and `Square` is a subclass of `Figure`. The call

```
screen.print( Square.new(10) );
```

would be illegal because there is no method

```
proc print( s : Square )
```

There should be used a cast:

```
screen.print( Figure.cast( Square.new(10) ) ) ;
```

The definition of subtype of Green prevents the compiler from choosing the nearest method to be used when the real parameter type does not match the formal parameter type of any overloaded method. To understand that, suppose methods

[4] But where the corresponding `new` methods would be added ? They could not be put in the public section of the class object ...

```
proc print( f : Figure )
proc print( w : Window )
```

belong to `MyScreen`. Class `Frame` is subtype of both `Figure` and `Window`. Now, which method should be called in the code below ?

```
var screen : MyScreen;
var f : Frame;

screen#init();
f#init();
...
screen.print(f);
```

If there was an order among all types, the compiler could choose the type nearest to `Frame`. But there may be no coherent or rational order — the definition of subtype is too loose to allow that. For example, `Figure` and `Window` may have the same set of method interfaces — the same type. In this case, which one should come first in the total ordering of the types ?

There may be a semantic error at run time if the class of the object `x` is a subtype (see Section 7.1) of its declared type:

```
var x : Figure;
x = Circle.new(20, 45);
screen.print(x);
```

In this case the compiler will generate code that will call method `print(Figure)` instead of `print(Circle)`.

7.7 Assertions and Subtyping

The observation on assertions and subclassing (Chapter 7.4) also applies to assertions and subtypes but with one big difference: it would be very difficult to enforce any rule about the relationship assertion/subtype method because the compiler does not know who are the supertypes of a class when compiling it. A class `A` is considered a supertype of `C` if `A` has at least all `C` method signatures. For the linker and compiler, however, a class `A` is supertype of `C` if:

- `C` is subclass of `A` or;
- there is an assignment
  `x = y`
  somewhere in the whole program such that the declared types of `x` and `y` are `A` and `C`, respectively or;
- `A` is a supertype of a class `B` which is supertype of `C`.

When one uses a type `A` in a variable declaration like
`var a : A;`
she expects the methods executed when sending messages to `a` to obey the assertions of `A` methods, even when variable `a` refers to a subtype object. To enforce this rule would be difficult because of the dynamic nature of Green which separates classes and types (and therefore objects and types since objects are created from classes and types are used to declare variables).

This problem could be solved if the compiler put a shell (see Chapter 12) in object `x` each time an assignment

```
x = exp;
```

were found and the declared type of `exp` were a subtype, but not a subclass, of the declared type of `x`. The shell would check the assertions of the methods. This assignment would be replaced by

```
x = exp;
Meta.attachShell( x, Assert_A.new() );
```

in which `A` is the declared class of `x` and `Assert_A` a shell class with the `class_A` assertions.

Variable `x` might be used, after this assignment, in a command

```
y = x;
```

in which the declared type of `y` is a supertype, but not a superclass, of the declared type of `x`. The the shell on `x` should be removed (should it ?) and another shell with class-of-`y` assertions should be attached to `x`.

The idea of putting shells to check assertions is very complex and would require the creation of great numbers of shell classes. Therefore this feature is not supported by Green.

Capítulo 8

The Basic Classes

All the basic types are classes with value semantics. This mean a variable of a basic type does not point to an object of that type. The variable contains the object of that type. Because of this, some restrictions applies to basic types: they cannot be subclassed and there is no polymorphism with them. A polymorphic variable of a type `T` may refer to objects of `T` and its subtypes, which may have different sizes. Then the variable should be a pointer which has a fixed size in bytes but can point to objects of any size. From here on, we will use "basic classes" for "basic types".

8.1 The Classes

The basic classes and their methods are shown below. `AnyValue` is superclass of all basic classes and neither `AnyValue` nor a basic class can be subclassed. `AnyValue`, like `Any`, does not inherit from any class. The basic classes use low level representation of characters, booleans, etc. These are the types of the machine to which the code is generated. For example, `lowLevelChar` is the machine type used to represent characters. Objects of basic classes cannot be dynamically allocated because they have value semantics. Albeit this limitation, a class object of a basic class is an object like any other. The arithmetic and logical operators are represented as methods in the following code although Green does not support operator overloading for normal classes. Note operator "=" for assignment is not a method of any class.

```
abstract class AnyValue
  public:
    abstract proc toString() : String
    abstract proc getInfo() : AnyObjectInfo
    abstract proc getClassInfo() : ClassInfo
    abstract proc getClassObject() : AnyClassObject
end


class char subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : char ) : boolean
      // returns true if self is equal to other

      /* these operators can only be used with variables. Unlike C++, they
```

```
        don't return anything. */
    proc ++()
    proc --()

      // comparison
    proc == ( other : char ) : boolean
    proc <> ( other : char ) : boolean
    proc <  ( other : char ) : boolean
    proc <= ( other : char ) : boolean
    proc >  ( other : char ) : boolean
    proc >= ( other : char ) : boolean

  private:
    var value : lowLevelChar;
end


class boolean subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : boolean ) : boolean
      // returns true if self is equal to other

    proc and( other : boolean ) : boolean
    proc or(  other : boolean ) : boolean
    proc xor( other : boolean ) : boolean
    proc not() : boolean

      // comparison
    proc == ( other : boolean ) : boolean
    proc <> ( other : boolean ) : boolean
    proc <  ( other : boolean ) : boolean
    proc <= ( other : boolean ) : boolean
    proc >  ( other : boolean ) : boolean
    proc >= ( other : boolean ) : boolean

  private:
    var value : lowLevelBoolean;
end


class byte subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : byte ) : boolean
      // returns true if self is equal to other

      /* these operators can only be used with variables. Unlike C++, they
```

```
        don't return anything. */
    proc ++()
    proc --()

      // unary
    proc +  () : byte
    proc -  () : byte
    proc ~  () : byte
      // binary
    proc +  ( other : byte ) : byte
    proc -  ( other : byte ) : byte
    proc *  ( other : byte ) : byte
    proc /  ( other : byte ) : byte
    proc %  ( other : byte ) : byte
    proc &  ( other : byte ) : byte
    proc |  ( other : byte ) : byte
    proc ^  ( other : byte ) : byte
    proc << ( other : integer ) : byte
    proc >> ( other : integer ) : byte

      // comparison
    proc == ( other : byte ) : boolean
    proc <> ( other : byte ) : boolean
    proc <  ( other : byte ) : boolean
    proc <= ( other : byte ) : boolean
    proc >  ( other : byte ) : boolean
    proc >= ( other : byte ) : boolean

  private:
    var value : lowLevelByte;
end


class integer subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : integer ) : boolean
      // returns true if self is equal to other

      /* these operators can only be used with variables. Unlike C++, they
        don't return anything. */
    proc ++()
    proc --()

      // unary
    proc +  () : integer
    proc -  () : integer
    proc ~  () : integer
```

```
      // binary
    proc +  ( other : integer ) : integer
    proc -  ( other : integer ) : integer
    proc *  ( other : integer ) : integer
    proc /  ( other : integer ) : integer
    proc %  ( other : integer ) : integer
    proc &  ( other : integer ) : integer
    proc |  ( other : integer ) : integer
    proc ^  ( other : integer ) : integer
    proc << ( other : integer ) : integer
    proc >> ( other : integer ) : integer

      // comparison
    proc == ( other : integer ) : boolean
    proc <> ( other : integer ) : boolean
    proc <  ( other : integer ) : boolean
    proc <= ( other : integer ) : boolean
    proc >  ( other : integer ) : boolean
    proc >= ( other : integer ) : boolean

     /* in the method that follow we assume an integer has at least
       four bytes */
    proc getByte1() : byte
      /* returns the value of the less significant byte of the integer. In some
         machines this will be the first byte. In others, it will be
         the last. The following methods work similarly. */
    proc getByte2() : byte
    proc getByte3() : byte
    proc getByte4() : byte

  private:
    var value : lowLevelInteger;
end


class long subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : long ) : boolean
      // returns true if self is equal to other

      /* these operators can only be used with variables. Unlike C++, they
         don't return anything. */
    proc ++()
    proc --()

      // unary
    proc +  () : long
```

```
    proc -  () : long
    proc ~  () : long
      // binary
    proc +  ( other : long ) : long
    proc -  ( other : long ) : long
    proc *  ( other : long ) : long
    proc /  ( other : long ) : long
    proc %  ( other : long ) : long
    proc &  ( other : long ) : long
    proc |  ( other : long ) : long
    proc ^  ( other : long ) : long
    proc << ( other : integer ) : long
    proc >> ( other : integer ) : long

      // comparison
    proc == ( other : long ) : boolean
    proc <> ( other : long ) : boolean
    proc <  ( other : long ) : boolean
    proc <= ( other : long ) : boolean
    proc >  ( other : long ) : boolean
    proc >= ( other : long ) : boolean

     /* in the method that follow we assume a long has at least
        eight bytes */
    proc getByte1() : byte
      /* returns the value of the less significant byte of the long. In some
         machines this will be the first byte. In others, it will be
         the last. The methods getByte2, getByte3, ... getByte8 work similarly. */


  private:
    var value : lowLevelLong;
end


class real subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : real ) : boolean
      /* returns true if self is equal to other. A warning should be
         issued if this method is used */

      /* these operators can only be used with variables. Unlike C++, they
         don't return anything. */
    proc ++()
    proc --()

      // unary
```

```
    proc +  () : real
    proc -  () : real
      // binary
    proc +  ( other : real ) : real
    proc -  ( other : real ) : real
    proc *  ( other : real ) : real
    proc /  ( other : real ) : real

      /* comparison. A warning should be issued whenever == or <> is used. */
    proc == ( other : real ) : boolean
    proc <> ( other : real ) : boolean
    proc <  ( other : real ) : boolean
    proc <= ( other : real ) : boolean
    proc >  ( other : real ) : boolean
    proc >= ( other : real ) : boolean

  private:
    var value : lowLevelReal;
end


class double subclassOf AnyValue
  public:
    proc toString() : String
    proc equals( other : double ) : boolean
      /* returns true if self is equal to other. A warning should be
         issued if this method is used */

      /* these operators can only be used with variables. Unlike C++, they
        don't return anything. */
    proc ++()
    proc --()

      /* unary
    proc +  () : double
    proc -  () : double
      /* binary */
    proc +  ( other : double ) : double
    proc -  ( other : double ) : double
    proc *  ( other : double ) : double
    proc /  ( other : double ) : double

      // comparison. A warning should be issued whenever == or <> is used.
    proc == ( other : double ) : boolean
    proc <> ( other : double ) : boolean
    proc <  ( other : double ) : boolean
    proc <= ( other : double ) : boolean
    proc >  ( other : double ) : boolean
```

```
    proc >= ( other : double ) : boolean

  private:
    var value : lowLevelDouble;
end



object char
  public:
    proc getSizeInBits() : integer
      // returns the size of a char in bits
    proc getSize() : integer
      // return the size of a char variable in bytes

    proc cast( value : String ) : char
    proc cast( any : AnyClass )
           ( exception : CatchTypeErrorException ) : char
      begin
      if not castOk(any) or the value of any cannot be converted to char
      then
        exception.throw( TypeErrorException.new() );
      endif
        // converts any to char
      ...
      end
    proc cast( value : byte ) : char
      assert
        before castOk(value);
      end
    proc cast( value : integer ) : char
      assert
        before castOk(value);
      end

    proc castOk( any : AnyClass ) : boolean
      begin
      var aClass : Any = any.getClassObject();
      return aClass == Char or aClass == Byte or
             aClass == Integer;
      end
    proc castOk( value : byte ) : boolean
      begin
      return value <= char.getMaxIntegerChar();
      end
    proc castOk( value : integer ) : boolean
      begin
      return value >= 0 and value <= char.getMaxIntegerChar();
```

```
      end


    proc getMinValue() : char
      begin
      return '\0';
      end
    proc getMaxValue() : char
      begin
      return '\x7F';
      end
    proc getMaxIntegerChar() : integer
        // returns maximum char in an integer value
      begin
      return 127;
      end
    proc getMinIntegerChar() : integer
      begin
      return 0;
      end
end


object boolean
  public:
    proc getSizeInBits() : integer
      // returns the size of a boolean in bits
    proc getSize() : integer
      // return the size of a boolean variable in bytes

    proc cast( value : String ) : boolean

    proc cast( any : AnyClass )
             ( exception : CatchTypeErrorException ) : boolean
      begin
      if not castOk(any) or the value of any cannot be converted to boolean
      then
        exception.throw( TypeErrorException.new() );
      endif
        // convert any to boolean
      ...
      end

    proc castOk( any : AnyClass ) : boolean
      begin
      var aClass : Any = any.getClassObject();
      return aClass == Boolean or aClass == Byte or
             aClass == Integer;
```

```
      end

    proc cast( value : integer ) : boolean
      // cast 0 to false and anything else to true

    proc cast( value : byte ) : boolean
      // cast 0 to false and anything else to true

    proc getMinValue() : boolean
      begin
      return false;
      end
    proc getMaxValue() : boolean
      begin
      return true;
      end
end


object byte
  public:
    proc getSizeInBits() : integer
      // returns the size of a byte in bits
    proc getSize() : integer
      // return the size of a byte variable in bytes


    proc cast( any : AnyClass )
             ( exception : TypeErrorException ) : byte
      begin
      if not castOk(any) or the value of any cannot be converted to byte
      then
        exception.throw( TypeErrorException.new() );
      endif
        // converts
      ...
      end

    proc cast( value : String ) : byte

    proc cast( value : boolean ) : byte
      assert
        before castOk(value);
      end

    proc cast( value : char ) : byte
      assert
        before castOk(value);
```

```
  end


proc cast( value : integer ) : byte
  assert
    before castOk(value);
  end

proc cast( value : long ) : byte
  assert
    before castOk(value);
  end

proc cast( value : real ) : byte
  assert
    before castOk(value);
  end

proc cast( value : double ) : byte
  assert
    before castOk(value);
  end

proc castOk( any : AnyClass ) : boolean
  begin
  var aClass : Any = any.getClassObject();
  return aClass == Boolean or aClass == Byte or
         aClass == Integer;
  end

proc castOk( value : integer ) : boolean
  begin
  return value >= byte.getMinValue() and
         value <= byte.getMaxValue();
  end

proc castOk( value : long ) : boolean
  begin
  return value >= byte.getMinValue() and
         value <= byte.getMaxValue();
  end

proc castOk( value : real ) : boolean
  begin
  return value >= byte.getMinValue() and
         value <= byte.getMaxValue();
  end
```

```
    proc castOk( value : double ) : boolean
      begin
      return value >= byte.getMinValue() and
             value <= byte.getMaxValue();
      end

    proc getMinValue() : byte
      begin
      return 0;
      end
    proc getMaxValue() : byte
      begin
      return 255;
      end
end


object integer
  public:
    proc getSizeInBits() : integer
      // returns the size of an integer in bits
    proc getSize() : integer
      // return the size of an integer variable in bytes

    proc cast( any : AnyClass )
             ( exception : TypeErrorException ) : integer
      begin
      if not castOk(any) or the value of any cannot be converted to integer
      then
        exception.throw( TypeErrorException.new() );
      endif
        // converts
      ...
      end

    proc cast( value : String ) : integer
    proc cast( value : char ) : integer
    proc cast( value : boolean ) : integer
    proc cast( value : byte ) : integer

    proc cast( value : long ) : integer
      assert
        before castOk(value);
      end

    proc cast( value : real ) : integer
      assert
        before castOk(value);
```

```
      end

    proc cast( value : double ) : integer
      assert
        before castOk(value);
      end

    proc castOk( any : AnyClass ) : integer
      begin
      var aClass : Any = any.getClassObject();
      return aClass == Char or aClass == Boolean or aClass == Byte or
             aClass == Integer or aClass == Long or
             aClass == Real or aClass == Double;
      end

    proc castOk( value : double ) : boolean
      begin
      return value >= integer.getMinValue() and
             value <= integer.getMaxValue();
      end

    proc castOk( value : real ) : boolean
      begin
      return value >= integer.getMinValue() and
             value <= integer.getMaxValue();
      end

    proc castOk( value : long ) : boolean
      begin
      return value >= integer.getMinValue() and
             value <= integer.getMaxValue();
      end

    proc getMinValue() : integer
      begin
      return -2147483647 - 1;
      end
    proc getMaxValue() : integer
      begin
      return 2147483647
      end
end


object long
  public:
    proc getSizeInBits() : integer
      // returns the size of a long in bits
```

```
proc getSize() : integer
  // return the size of a long variable in bytes

proc cast( any : AnyClass )
         ( exception : TypeErrorException ) : long
  begin
  if not castOk(any) or the value of any cannot be converted to long
  then
    exception.throw( TypeErrorException.new() );
  endif
    // converts
  ...
  end

proc cast( value : String ) : long
proc cast( value : byte ) : long
proc cast( value : integer ) : long
proc cast( value : real ) : long
  assert
    before castOk(value);
  end
proc cast( value : double ) : long
  assert
    before castOk(value);
  end

proc castOk( any : AnyClass ) : boolean
  begin
  var aClass : Any = any.getClassObject();
  return aClass == Long or aClass == Integer or aClass == Byte
         aClass == Real or aClass == Double;
  end

proc castOk( value : double ) : boolean
  begin
  return value >= long.getMinValue() and
         value <= long.getMaxValue();
  end

proc castOk( value : real ) : boolean
  begin
  return value >= long.getMinValue() and
         value <= long.getMaxValue();
  end

proc getMinValue() : long
  begin
  return -2147483647*2147483647 - 1;
```

```
      end
    proc getMaxValue() : long
      begin
      return 2147483647*2147483647;
      end
end


object real
  public:
    proc getSizeInBits() : integer
      // returns the size of a real in bits
    proc getSize() : integer
      // return the size of a real variable in bytes

    proc cast( any : AnyClass )
             ( exception : TypeErrorException ) : real
      begin
      if not castOk(any) or the value of any cannot be converted to real
      then
        exception.throw( TypeErrorException.new() );
      endif
        // converts
      ...
      end

    proc cast( value : String ) : real
    proc cast( value : byte ) : real
    proc cast( value : integer ) : real
    proc cast( value : long ) : real
    proc cast( value : double ) : real
      assert
        before castOk(value);
      end

    proc castOk( any : AnyClass ) : boolean
      begin
      var aClass : Any = any.getClassObject();
      return aClass == Byte or aClass == Integer or aClass == Long
             or aClass == Real or aClass == Double;
      end

    proc castOk( value : double ) : boolean
      begin
        // platform-dependent code
      end
```

```
    /* each of the following methods is based on a similar constant
       found in the header file "limits.h" of the standard C
       library. The method description is a  copy of the
       description of the corresponding constant as found in K&R. The
       method bodies are dependent on the platform and therefore not
       defined. */

  proc getRadix() : integer
    // radix of exponent representation
  proc getRounds() : integer
    // mode for addition
  proc getPrecision() : integer
    // decimal digits of precision
  proc getEpsilon() : real
    // smaller number e such that 1.0 + e <> 1.0
  proc getMantDig() : integer
    // number of base ‘‘getRadix()’’ in mantissa
  proc getMinValue() : real
    // minimum real number
  proc getMaxValue() : real
    // maximum real number
  proc getMaxExp() : real
    // greater k such that k^n is representable
  proc getMinExp() : real
    // smaller k such that k^n is a normalized number

end
```

```
object double
 public:
  proc getSizeInBits() : integer
    // returns the size of a double in bits
  proc getSize() : integer
    // return the size of a double variable in bytes

  proc cast( any : AnyClass )
           ( exception : TypeErrorException ) : double
    begin
    if not castOk(any) or the value of any cannot be converted to double
    then
      exception.throw( TypeErrorException.new() );
    endif
      // converts
    ...
    end
```

```
    proc cast( value : String ) : double
    proc cast( value : byte ) : double
    proc cast( value : integer ) : double
    proc cast( value : long ) : double
    proc cast( value : real ) : double

    proc castOk( any : AnyClass ) : double
      begin
      var aClass : Any = any.getClassObject();
      return aClass == Byte or aClass == Integer or aClass == Long or
             aClass == Real or aClass == Double;
      end

      /* each of the following methods is based on a similar constant
         found in the header file "limits.h" of the standard C
         library. The method description is a  copy of the
         description of the corresponding constant as found in K&R. The
         method bodies are dependent on the platform and therefore not
         defined. */

    proc getRadix() : integer
      // radix of exponent representation
    proc getRounds() : integer
      // mode for addition
    proc getPrecision() : integer
      // decimal digits of precision
    proc getEpsilon() : double
      // smaller number e such that 1.0 + e <> 1.0
    proc getMantDig() : integer
      // number of base ‘‘getRadix()’’ in mantissa
    proc getMinValue() : double
      // minimum double number
    proc getMaxValue() : double
      // maximum double number
    proc getMaxExp() : double
      // greater k such that k^n is representable
    proc getMinExp() : double
      // smaller k such that k^n is a normalized number
end
```

Notice that

- an object can be cast to a basic class through methods like
```
     proc cast( any : AnyClass )
              ( exception : TypeErrorException ) : char
```
  of class object `char`. This will only succeed if the class of `any` at run time is `Char`, `Byte`, or `Integer` and the value can be converted to a character;

- every basic value can be transformed into a string:

```
s = 5.toString();          // s will point to "5"
s = #'A'.toString();       // s will point to "A"
```

- `real` and `double` numbers are transformed into strings using the format `[-]m.ddddddE+-xx` where `m` is the number before the decimal point;
- the assertions used in the basic class definitions make use of some implicit conversions such as from `integer` to `long` in method `castOk( value : long )` of class object `integer`;
- hopefully the compiler will able to enable/disable the assertions for basic class methods. If an assertion of a `cast` method is violated, the run-time system will throw an exception of one of the following subclasses of `AssertionException`:

```
AssertionCastCharException
AssertionCastBooleanException
AssertionCastByteException
AssertionCastIntegerException
AssertionCastLongException
AssertionCastRealException
AssertionCastDoubleException
```

Class `AssertionCastCharException` has methods

```
proc init( originalValueClass : AnyClassObject; value : Any )
proc getOriginalValueClass() : AnyClassObject;
proc getOriginalValue() : Any
```

`originalValueClass` is the class of the object that could not have been converted to `char`. `value` is transformed into an object of a wrapper class (see next Section). To understand this better, consider the code

```
var i : integer = 300;
var ch : char;
ch = char.cast(i);  // throws exception
```

At run time, exception `AssertionCastCharException` will be thrown by a call

```
exception.throw( AssertionCastCharException.new( integer,
                                                 Integer.new(i) ) )
```

Class `Integer` is a wrapper class described next.

8.2 Wrapper Classes

There are classes `Char`, `Boolean`, `Byte`, `Integer`, `Long`, `Real`, and `Double` that obey the reference semantics and mirror the basic classes. Of course, all inherit from `AnyClass`. Automatic conversion is provided between an object of one of these classes and its corresponding basic class value. So, the code below is legal.

```
    var I : Integer;
        i : integer;
begin
i = 1;
I = 1;
i = I;
I = i;
I = Integer.new(5);
i = Integer.new(5);
i = i + I;
i = 6*I + 10 - I;
I = 3*i*I;
end
```

The compiler inserts code to create objects of class `Integer` and to convert these objects in `integer` values. For example, the statements

```
I = 1;
i = 5*I;
```

would be implemented as

```
I = Integer.new(1);
i = 5*I.get();
```

Classes like `Integer` are called wrapper classes because they pack basic values in objects. Since class `Integer` and the like are subclasses of `Any`, they allow the program to treat basic values as objects. For example, a method

```
proc print( any : Any )
  begin
  Out.write( any.toString() );
  end
```

accepts values of any basic class and any object as parameter. So,

```
a.print(5);
```

is transformed into

```
a.print( Integer.new(5) );
```

Whenever a basic class value is used where an object is expected, a wrapper object is automatically created.

The automatic conversion between basic classes and wrapper classes is one of the novelties of Green. This allows one to see everything in the program as objects, a feature supported only by very dynamic languages as Smalltalk and Self. On the other hand, the programmer can use the basic classes for better performance and convert basic values to objects when necessary. There is only one restriction on wrapper classes: they cannot be subclassed. This is necessary to allow the compiler to optimize their use.

Wrapper classes add a restriction on overloaded methods: a class cannot have two methods

```
proc print( i : integer )
proc print( i : Integer )
```

There would be an ambiguity in a call

```
s.print(5)
```

because 5 is converted to Integer whenever necessary. In the general case, a class has illegal method declarations if, after changing the type of all parameters to the corresponding wrapper classes, two methods have the same name, number of parameters, and parameter types.

The Char class is shown next. The other wrapper classes are similar. After a wrapper object is created, its value is never modified.

```
class Char
    proc init( value : char )
      begin
      self.value = value;
      end
  public:
    proc get() : char
      begin
      return value;

      end
  private:
    var value : char;
end
```

The ++ operator increments its operand and can be used with a wrapper variable of any class but Boolean. After the application of ++ on a variable, it will refer to a different object since wrapper objects never change its value. So, the code

```
var I : Integer;
I = 1;
++I;
```

creates two wrapper objects.

Capítulo 9

Strings

Any sequence of characters between " is an object of type `String`, a predefined class. The character " itself can be put in the sequence by prefixing it with \. Examples of `String` handling are given below.

```
var s : String = "this is a string";
var p : String;

p = s + " yet " + "another" + " string " + "(YAS)" + "\n";
```

The operator `+` is overloaded to concatenate two string objects returning a *new* string and `\n` is the new line character.

A string can be initialized in its creation:

```
var s, t : String;
s = String.new("a string");
t = String.new(hour, "h ", min, "min ", "am");
```

If `hour = 8`, `min = 35`, `t` would hold the string `"8h 35min am"`. A literal string such as "a string" is a compiler allocated object.

For performance sake, class `String` cannot be subclassed, has no subtypes, and does not provide any method for modifying its content. If you need to change a string you should use objects of class `DynString`. Then a variable of class `String` will always point to a string object that is never modified. The definitions of classes `String` and `DynString` are given below.

```
object String
   public:
      proc cast( x : char ) : String
      proc cast( x : boolean ) : String
      proc cast( x : byte ) : String
      proc cast( x : integer ) : String
      proc cast( x : long ) : String
      proc cast( x : real ) : String
      proc cast( x : double ) : String
end

class String
    proc init( s : String )
```

```
    // initializes self with s
    assert
      before s <> nil;
    end

  proc init( v : ...array(Any)[] )
    // initializes self with strings corresponding to v values

public:
  proc get( i : integer ) : char
      // get character at position i
    assert
      before i >= 0 and i < getSize();
    end

  proc getIter() : DS.Iter(char)
    // returns an iterator for the string. See Chapter "The Standard Library"

  proc cmp( other : String ) : integer
    /* compare strings self and other. Returns -1, 0, or 1 if self
       is less than, equal to, or greater than other, respectively.
       This is the same as the C strcmp function. */
    assert
      before other <> nil;
    end

  proc cmpIgnoreCase( other : String ) : integer
    /* the same as cmp but ignoring differences between upper and
       lower case letters */
    assert
      before other <> nil;
    end

  proc newConcat( other : String )
              ( exception : CatchOutOfMemoryException ) : String
    /* creates and returns a new String that is the concatenation of
       self with other */
    assert
      before other <> nil;
    end

  proc tocharArray( copyto : array(char)[] )
                ( exception : CatchOutOfMemoryException )
    // copies string into array copyto
    assert
      before copyto.getSize() >= getSize();
    end
```

```
proc tocharArray( copyto : array(char)[]; i : integer )
                ( exception : CatchOutOfMemoryException )
  /* copies string into array copyto beginning at position
     i of copyto */
  assert
    before copyto.getSize() - i >= getSize();
  end

proc tobyteArray( copyto : array(byte)[] )
                ( exception : CatchOutOfMemoryException )
  // copies string into array copyto
  assert
    before copyto.getSize() >= getSize();
  end

proc tobyteArray( copyto : array(byte)[]; i : integer )
                ( exception : CatchOutOfMemoryException )
  /* copies string into array copyto beginning at position
     i of copyto */
  assert
    before copyto.getSize() - i >= getSize();
  end

proc getSize() : integer
  // return the number of elements of the string

proc newToLowerCase()
                   ( exception : CatchOutOfMemoryException ) : String
  // returns a new string with all letters changed to lowercase

proc newToUpperCase()
                   ( exception : CatchOutOfMemoryException ) : String
  // returns a new string with all letters changed to uppercase

proc getSubset( from, to2 : integer )
              ( exception : CatchOutOfMemoryException ) : String
  /* returns a new string with all characters of this string
     between positions from and to2. to is a reserved word, then to2. */
  assert
    before from >= to2 and 0 <= from and to2 < getSize;
    after  result.getSize() == to2 - from + 1;
  end

proc search( s : String ) : integer
  /* searchs for string s in self.  Returns the index of s in self
     or -1 if it was not found. */
  assert
    before s <> nil;
```

```
        after  result >= -1 and result < getSize();
      end

    proc hashCode() : integer

    proc tobyte() : byte
      /* assumes the string is a number and converts it to a byte.
         The following methods work similarly. */
      assert
        before tobyteOk();
      end

    proc tointeger() : integer
      assert
        before tointegerOk();
      end

    proc tolong() : long
      assert
        before tolongOk();
      end

    proc toreal() : real
      assert
        before torealOk();
      end

    proc todouble() : double
      assert
        before todoubleOk();
      end

    proc toDynString()
                    ( exception : CatchOutOfMemoryException ) : DynString

end


class DynString
    proc init( s : String )
      // initializes self with s
      assert
        before s <> nil;
      end

    proc init( s : DynString )
      // initializes self with s
      assert
```

```
    before s <> nil;
  end

public:

  proc get( i : integer ) : char
      // get character at position i
    assert
      before i >= 0 and i < getSize();
    end

  proc getIter() : DS.Iter(char)
    // returns an iterator for the string. See Chapter "The Standard Library"

  proc cmp( other : DynString ) : integer
    /* compare strings self and other. Returns -1, 0, or 1 if self
       is less than, equal to, or greater than other, respectively.
       This is the same as the C strcmp function. */
    assert
      before other <> nil;
    end

  proc cmpIgnoreCase( other : DynString ) : integer
    /* the same as cmp but ignoring differences between upper and
       lower case letters */
    assert
      before other <> nil;
    end

  proc concat( other : String )
           ( exception : CatchOutOfMemoryException )
      /*  creates and returns a new DynString that is the concatenation of
          self with other */
    assert
      before other <> nil;
    end

  proc tocharArray( copyto : array(char)[] )
                 ( exception : CatchOutOfMemoryException )
    // copies string into array copyto
    assert
      before copyto.getSize() >= getSize();
    end

  proc tocharArray( copyto : array(char)[]; i : integer )
                 ( exception : CatchOutOfMemoryException )
    /* copies string into array copyto beginning at position
       i of copyto */
```

```
  assert
    before copyto.getSize() - i >= getSize();
  end

proc tobyteArray( copyto : array(byte)[] )
                ( exception : CatchOutOfMemoryException )
  // copies string into array copyto
  assert
    before copyto.getSize() >= getSize();
  end

proc tobyteArray( copyto : array(byte)[]; i : integer )
                ( exception : CatchOutOfMemoryException )
    /* copies string into array copyto beginning at position
       i of copyto */
  assert
    before copyto.getSize() - i >= getSize();
  end

proc getSize() : integer
  // return the number of elements of the string

proc toLowerCase()
                ( exception : CatchOutOfMemoryException )
  // returns a new string with all letters changed to lowercase

proc toUpperCase()
                ( exception : CatchOutOfMemoryException )
  // returns a new string with all letters changed to uppercase

proc getSubset( from, to2 : integer )
              ( exception : CatchOutOfMemoryException ) : DynString
    /* returns a new string with all characters of this string
       between positions from and to2. to is a reserved word, then to2. */
  assert
    before from >= to2 and 0 <= from and to2 < getSize;
    after  result.getSize() == to2 - from + 1;
  end

proc search( s : DynString ) : integer
    /* searchs for string s in self.  Returns the index of s in self
       or -1 if it was not found. */
  assert
    before s <> nil;
    after  result >= -1 and result < getSize();
  end

proc hashCode() : integer
```

```
proc tobyte() : byte
  assert
    before tobyteOk();
  end

proc tointeger() : integer
  assert
    before tointegerOk();
  end

proc tolong() : long
  assert
    before tolongOk();
  end

proc toreal() : real
  assert
    before torealOk();
  end

proc todouble() : double
  assert
    before todoubleOk();
  end

proc removeSpaceBegin()
  /* removes all spaces of the beginning of the string. A space is
     one of the followings: ' ', '\t', '\r', '\n' . */

proc removeSpaceEnd()
  /* removes all spaces of the end of the string. A space is
     one of the followings: ' ', '\t', '\r', '\n' . */

proc toString()
             ( exception : CatchOutOfMemoryException )  : String
  // returns a String corresponding to this DynString

proc prepend( toadd : DynString )
            ( exception : CatchOutOfMemoryException )
  // add toadd at the beginning of self

proc removeAllCh( ch : char ) : boolean
  // removes all characters ch of the string

proc remove( i : integer )
  // removes the character i of the string
  assert
```

```
        before i >= 0 and i < getSize();
      end

    proc insert( i : integer; ch : char )
               ( exception : CatchOutOfMemoryException )
      /* inserts character ch at position i of the string. The previous
         character at this position is right shifted */
      assert
        before i >= 0 and i < getSize();
      end

    proc add( i : integer; ch : char )
            ( exception : CatchOutOfMemoryException )
      // replaces character of position i by ch
      assert
        before i >= 0 and i <= getSize();
      end


    proc add( ch : char )
            ( exception : CatchOutOfMemoryException )
      // adds ch at the end of the string
end
```

Capítulo 10

Class Objects and Subtyping

10.1 Types and Class Objects

A class object defines an object at compile time. A class specifies the instance variables/methods its objects will have at run time. Class object `Window` of Figure 10.1 *defines* the object `Window` and as such it can assign a value to variable `defaultColor` in the line

```
var defaultColor : integer = 12;
```

This assignment would be illegal if it were in class `Window` since a class declaration is a *type declaration*. It only specifies the shape of the objects of that class and could not have an action such as an assignment attached to it.

Although class objects do not have a class, they do have a type, which is just the set of public method signatures of the object. Then, the type of class object `Window` is

```
{ getDefaultColor() : integer,
  new() : Window,
  ...
  }
```

The dots (...) represent some methods the compiler adds to each class object. These methods will be presented in Section 10.3. Method `new` is added by the compiler since class `Window` defines a method `init`.

Now we are ready to define the compile-time function `type`. This function takes a variable as a parameter, which includes classes.[1] Function `type` is defined as follows.

- `type(a)` is the declared type of `a` if `a` is a user-defined variable. For example,

  ```
  var  x : integer;
       y : type(x);
       a : A;
       b : type(a);
  ```

  declares the type of `y` as `integer` and the type of `b` as `A`. The statement

  ```
  var a : A = type(a).new()
  ```

  is equivalent to

  ```
  var a : A = A.new();
  ```

[1] If there is a class `A`, the program has a variable `A` that points to the class object corresponding to this class.

```
object Window
  public:
    proc getDefaultColor() : integer
      begin
      return defaultColor;
      end
  private:
    var defaultColor : integer = 12;
end

class Window
    proc init()
      begin
      ...
      end

  public:
    ...
end
```

Figura 10.1: Class object and class `Window`

- `type(A)` is the set of methods of class object `A` if `A` is a class name. Then,

  ```
  var W : type(Window);
  ```

  declares a variable `W` to which can be assigned an object whose type is a subtype of `type(Window)`.

  To `W` may be assigned `Window`:

  ```
  W = Window;
  i = W.getDefaultColor();  // calls method of Window
  ```

  In fact, the compiler will create an abstract class with method signatures equal to those of `Window`. This class will replace "`type(Window)`".

If a class defines constants or enumerated constants in its public section, these do not belong to the class type since a type is just a set of *method* interfaces.

10.2 Delegation and Class Objects

Since class objects are just objects, there is no inheritance relationship among them. Sharing of methods among class objects is made through a delegation mechanism. For example, class object `B` of Figure 10.2 delegates the `m` message to class `A`. Method `m` of `B` should be called as

```
B.m(B)
```

that will execute

```
A.m(B)
```

that will call method `n` of parameter `a` in

```
a.n() + 1
```

Since `a` refer to class object `B`, this will call method `n` of `B`.

```
abstract class TypeA
  public:
    abstract proc m( x : TypeA ) : integer
    abstract proc n() : integer
end

object A
  public:
    proc m( a : TypeA ) : integer
      begin
      return a.n() + 1;
      end
    proc n() : integer
      begin
      return 0;
      end
end

object B
  public:
    proc m( a : TypeA ) : integer
      begin
      return A.m(a);
      end
    proc n() : integer
      begin
      return 1;
      end
end
```

Figura 10.2: Delegation using class objects

```
abstract class TypeA
  public:
    abstract proc m(x : TypeA) : integer
    abstract proc n() : integer
end

object A
  public:
    proc m() : integer
      begin
      return n() + 1;
      end
    proc n() : integer
      begin
      return 0;
      end
end

object B
  public:
    proc m() : integer
      begin
      return A.m();
      end
    proc n() : integer
      begin
      return 1;
      end
end
```

Figura 10.3: Forwarding of messages using class objects

```
object Any
  public:
    proc equals( other : Any ) : boolean
    proc toString( p : Any ) : String
    proc isObjectOf( p : Any; aClass : AnyClassObject ) : boolean
    proc getInfo( p : Any ) : AnyObjectInfo
    proc shallowClone( p : Any ) : Any;
    proc deepClone   ( p : Any ) : Any;
    proc shallowCopy ( p : Any; other : Any ) : boolean;
    proc shallowEqual( p : Any; other : Any ) : boolean;
    proc deepEqual   ( p : Any; other : Any ) : boolean;
      // there are other methods
    ...
end
```

Figura 10.4: Class object `Any`

This means method `m` of `A` knows the identity of the original receiver of message `m`, which was `B`. This is the true delegation. If `m` of `A` did not know the original receiver, there would be `forwarding` of messages, as is shown in Figure 10.3.

In this case,

```
B.m();
```

will call method `m` of `A` that will call method `n` of `A`.

10.3 Methods Added to Class Objects

We have seen that class `Any` defines the methods

- `getInfo() : AnyObjectInfo`
- `equals( other : Any ) : boolean`
- `toString() : String`
- `isObjectOf( aClass : AnyClassObject ) : boolean`
- `shallowClone() : Any`
  `( exception : CatchOutOfMemoryException )`
- `deepClone() : Any`
  `( exception : CatchOutOfMemoryException )`
- `shallowCopy( other : Any ) : boolean`
- `shallowEqual( other : Any ) : boolean`
- `deepEqual( other : Any ) : boolean`

that are inherited by all classes. Each of these methods calls a similar method in class object `Any`. As an example, method `shallowClone` of class `Any` is defined as

```
class Any
  public:
    proc shallowClone() : Any;
      begin
      return Any.shallowClone(self);
      end
    ...
end
```

These `Any` methods are then inherited by all classes. Note `Any` of the expression

`Any.shallowClone(self)`

refers to the *class object* `Any`. Class object `Any` defines a method with an extra parameter

`p : Any`

for each of the methods of class `Any` listed above — see Figure 10.4.

To each *class object* are added methods with the same signatures as those of class `Any`. These methods are defined as follows.

```
object Person
  public:
    proc shallowClone() : Any
      begin
      return Any.shallowClone(Person);
      end
    ...
end
```

By the above, class object `Any` also has a method

`proc shallowClone() : Any`

In a message send

`a.shallowClone()`

where `a` points to a class-`A` object, method `shallowClone()` of class `Any` is called. This method calls method

`shallowClone(Any) : Any`

of class object `Any` in the statement

`return Any.shallowClone(self);`

Pseudo-variable `self`, which points to the same object as `a`, is passed as a parameter. If this method of class object `Any` needs to call a method of the object being cloned, it will call it by sending a message to its first parameter. Then the search for the method will begin in object `a`. As we have seen, this is true delegation.

The statement

`p = Person.shallowClone();`

calls method `shallowClone` of class object `Person` that executes

`Any.shallowClone(Person)`

creating a new class.

Class object `Any` defines a method

```
proc basicNew( aClass : AnyClassObject )
           ( exception : CatchCreationException ) : AnyClass
```

which takes a class object as parameter an returns an uninitialized object of that class. Exception

`CreationException` is thrown if `aClass` is not a class object[2] or if no object of that class can be created. This last case occur if `aClass` is an abstract class or if it is not visible by some reason (this may occur if private classes are to be supported by Green).

An example of use of `basicNew` could be

```
var a : Account;
a = Any.basicNew(Account);
a.set(myName, myBank, myBalance);
```

`basicNew` creates an object but it does not initialize it — no `init` method of `Account` is called. Then the object instance variables should be initialized by other methods, as `set` in the above example.

`basicnew` may be used to create a singleton class:

```
object Earth
  proc init()
    begin
    try(HCatchAll)
      earth = Any.basicNew(Earth);
    end
    end
  public:
    proc new() : Earth
      begin
        // always return the same object
      return earth;
      end
  private:
    var earth : Earth;
end

class Earth subclassOf Planet
    // no init methods
  ...
end
```

To each class object `A` the compiler adds methods

- ```
  proc cast( x : Any )
           ( exception : CatchTypeErrorException ) : A
    begin
    if x.isObjectOf(A)
    then
      return x;   // converts x to A
    else
      exception.throw( TypeErrorException.new() );
    endif
    end
  ```

  that casts object `x` into type `A`. It throws an exception `TypeErrorException` if the `x` class is not a subtype of `A`.

[2] It may be a subtype of `AnyClassObject` without being a class object.
  ```

- `getAssociateClassInfo() : ClassInfo`

  returns information about the class associated to this class object.

- `getInitMethod() : ObjectInitMethodInfo`

  returns information on the `init` method of the class object.

- ```
  castObject( any : Any )
       ( exception : CatchTypeErrorException ) : type(A)
  ```
  returns the object `any` cast into type `type(A)`. Exception `CatchTypeErrorException` is thrown if this is not possible.

The last two methods are also defined in the abstract class `AnyClassObject` that inherits from `Any`. A variable of type `AnyClassObject` can refer to any class object. `AnyClassObject` is *supertype* of the type of any class object.

The definition of class objects and of class `Any` implies that:

- each class inherits from `Any` and is therefore a subtype of it;

- to each class object are added methods with the same signatures as those of class `Any`. Therefore, the type of each class object is a subtype of the type of `Any`.

We conclude that the type of each object, whatever it is, is a subtype of the type of class `Any`. Note the definition of type was extended to cope with objects and not only classes.

If one wants to modify a method automatically added to a class object as `shallowClone`, she should define this method in the class object by herself:

```
object MyWindow
  public:
    proc shallowClone() : Any
      begin
        // my code
        ...
      end
end
```

Then the compiler will not add a `shallowClone` method to this class object.

If one wants to redefine all `shallowClone` methods of all class objects, she should attach a shell (Chapter 12) with a method

```
proc shallowClone( p : Any ) : Any
```

to class object `Any`. This is the Green way of getting the functionality of other languages that support metaclasses. In one of these languages, there could be a metaclass `MetaClass` from which all metaclasses inherit. In a language like this, one could redefine all `shallowClone` methods of all class objects by changing method `shallowClone` of `MetaClass`.
```

Capítulo 11

Exceptions

11.1 Definition

Suppose one wants to open, read, and close a text file with a class `File`:

```
var f : File;
var text : array(char)[];

f = File.new();
f.open("mytext");
text = array(char)[].new( f.getSize() );
f.read(text);
f.close();
```

There may be some error when allocating memory, opening, reading, or closing the file. In languages without an exception system, there are two main patterns of taking care of these errors:

1. requiring every method call that may fail to return a value (e.g. `false`) showing it failed;

2. requiring every method call that may fail to set a global variable with the error number.

In any case the programmer is forced to fill the code with a lot of `if` statements to test for errors. This makes the code obscure because code for two different jobs are mixed statement by statement: reading a file and treating exceptional conditions.

This code can be rewritten as

```
var f : File;
var text : array(char)[];
var errorCatch : CatchFileException;

errorCatch = CatchFileException.new();
try(errorCatch)
    f = File.new();
    f.open("mytext");
    text = array(char)[].new( f.getSize() );
    f.read(text);
    f.close();
end
```

Now any error signaled by the code between `try` and `end` is handled by the methods of class `CatchFileException`. The methods `new`, `open`, `read`, and `close` throw an error by sending a message `throw` to object `exception`:

```
exception.throw( FileOpenException.new(fileName) );
```

`FileOpenException` is a direct or indirect subclass of class `Exception`. When `exception` receives a message `throw`, the run-time system looks for a `try` clause in the stack of called methods that can handle this exception. In the stack of called methods there may be several methods with `try` statements, each one with an object like `errorCatch` of the previous example. The run-time system will send a message `throw` to the object of one of the `try` statements and remove all methods of the stack of called methods to this point. Then all methods that could not have handled the exception are terminated. Now there is a question: how the run-time system chooses the `try` statement to be used ? To explain that, suppose the stack of called methods is m_1, m_2, ... m_k of which m_k is at the top. Each of these methods may have one or more `try` statements (maybe nested) in such a way that

c_1, c_2, ... c_n

is the stack of objects of the `try` statements. Each time the execution reaches a `try` statement its object is pushed into the stack. When the `end` of the `try` statement is reached the object is removed from the stack.

When `exception` receives a message

```
exception.throw( exceptionObject )
```

the run-time system looks in the `try` stack for an object that has a method

```
proc throw( p : E )
```

in such a way the type of object `exceptionObject` is a subtype of `E`.[1] Suppose the object found is c_i, $1 \leq i \leq n$. Then all methods corresponding to the `try` statements of the objects c_{i+1}, ... c_n are terminated and the statement

c_i`.throw(exceptionObject)`

is executed. After this the execution continues after the `end` of the `try` statement of c_i, unless the above statement also raises an exception.

Before discussing further details of exceptions, let us complete the initial example. Class `CatchFileException` defines `throw` methods for handling exceptions `FileOpenException`, `FileReadException`, and `FileCloseException`. It is given below together with class `FileOpenException` and a skeleton of method `open` of `File`.

```
class CatchFileException subclassOf CatchUnchecked
  proc init()
    begin
    end

  public:

    proc throw( exc : FileOpenException )
      begin
      Out.writeln( "Cannot open file ", exc.getFileName() );
      Runtime.exit(1);
      end

    proc throw( exc : FileReadException )
```

[1] Types of objects are defined similarly to types of classes. See Chapter 10.

```
      begin
      Out.writeln( "Error in reading the file ", exc.getFileName() );
      Runtime.exit(1);
      end

    proc throw( exc : FileCloseException )
      begin
      Out.writeln( "Error in closing the file ", exc.getFileName() );
      Runtime.exit(1);
      end
end


class File
  public:
      // for a while ignore the exception parameter
    proc open( fileName : String )
             ( exception : CatchFileException )
      begin
      ...
      exception.throw( FileOpenException.new(fileName) );
      ...
      end
    ...
end

class FileOpenException subclassOf Exception
  public:
    proc init( pfileName : String )
      begin
      fileName = pfileName;
      end
    proc getFileName() : String
      begin
      return fileName;
      end
  private:
    var fileName : String;
end
```

The initial example is

```
var f : File;
var text : array(char)[];
var errorCatch : CatchFileError;

errorCatch = CatchFileError.new();
try(errorCatch)
  f = File.new();
```

```
  f.open("mytext");
    // the same as text = array(char)[].new(f.getSize())
  text#init( f.getSize() );
  f.read(text);
  f.close();
end
```

Suppose there is an error in method `open` of `File` called by statement

```
f.open("mytext");
```

and an exception is thrown by statement

```
exception.throw( FileOpenException.new(fileName) );
```

The execution of method `File::open` is terminated and the run-time system looks for a `throw` method in object `errorCatch` since the call to `open` is inside a "`try(errorCatch) ... end`" block. Variable `errorCatch` refers to an object of class `CatchFileError` that does have a method

```
proc throw( exc : FileOpenException )
```

Then the statement

```
errorCatch.throw( exceptionObject )
```

is executed in which `exceptionObject` refers to the object "`FileOpenException.new(fileName)`" created in the message send

```
exception.throw( FileOpenException.new(fileName) );
```

in `File::open`.

Although exceptions may look complex, they in fact are not if we take another point of view: sending a message to object `exception` is to send the same message to an object of a `try` statement. In fact, the first object in a stack of try objects that can handle the message.

11.2 Hierarchy of Exceptions

Now it is time to fill some gaps in the definition of exceptions. First, every exception class like `FileOpenException` must be *subtype* (not necessarily subclass) of class `Exception`. This class defines only method `toString` which is supposed to return a string describing the exception.

There are some predefined exception classes in Green. They are arranged next in such a way a subclass is two columns ahead of its superclass of the previous line. Subclasses of the same class are in the same column. Then, `StackOverflowException` and `IllegalArrayIndexException` are subclasses of `UncheckedException`.

```
Exception
  TypeErrorException
  WrongParametersException
  NotFoundException
  PackedException
  TooManyDimensionsException

  MetaException
    ClassNotInAllowedSetException
    NoShellException
    NoExtensionException

  UncheckedException
    StackOverflowException
```

```
    IllegalArrayIndexException
    OutOfMemoryException
    InternalErrorException
    MessageSendToNilException

    NoReflectiveInfoException
      NoReflectiveBodyInfoException
      NoReflectiveCallInfoException

    ArithmeticException
      DivisionByZeroException
      RealOverflowException
      RealUnderflowException

    AssertionException
      AssertionAfterException
      AssertionBeforeException
      AssertionCastCharException
      AssertionCastBooleanException
      AssertionCastByteException
      AssertionCastIntegerException
      AssertionCastLongException
      AssertionCastRealException
      AssertionCastDoubleException
```

User exception classes are usually direct subclasses of `Exception`. Subclasses of `UncheckedException` are used by the compiler to signal some errors that programmers usually do not care about.

11.3 Typing Exceptions

Each method must declare the exceptions it can throw as the type of the parameter `exception` as in method `Fill::open`:

```
proc open( fileName : String )
         ( exception : CatchFileException )
```

Here `CatchFileException` can be replaced by the more specific class `CatchFileOpenException`. To each direct or indirect subclass of `Exception`, say `AnError`, the Green compiler will automatically create classes

```
class CatchAnError subclassOf CatchUncheckedException
  public:
    proc init()
      begin
      end

    proc throw( exc : AnError )
      begin
      end
end
```

```
class HCatchAnError subclassOf CatchUncheckedException
  public:
    proc init()
      begin
      end
    proc throw( exc : AnError )
      begin
      Out.writeln("Exception AnError not caught");
      Runtime.exit(1);
      end
end
```

that can be readily used as the `exception` type. The last class is a *hard* catch exception class. If an object of this class catches an exception it terminates the program.

Class `CatchUncheckedException` is a predefined class with a `throw` method for each subclass of `UncheckedException`. Each of these methods just throws the exception again:

```
class CatchUncheckedException subclassOf Catch
   public:
     proc throw( exc : StackOverflowException )
               ( exception : CatchUncheckedException )
       begin
       exception.throw(exc);
       end
     ...
end
```

Class `HCatchUncheckedException` inherits from class `Catch` (Figure 11.1) and defines a method `throw` for each subclass of `UncheckedException`. Each of these methods prints an explaining message in the default output device and exits the program. Class `CatchUncheckedException` must be supertype of every catch class or catch class object. A catch class is a class whose objects are used as arguments to `try` commands. A catch class object is a class object used as argument to a `try` command. Objects of class `Catch` keep a reference to the exception object that was thrown in a statement

```
     exception.throw( exceptionObject );
```

It also keeps a reference to the class of the object as considered by the run-time system, which may be different from

```
     exceptionObject.getClassObject()
```

See details ahead.

The run-time system may throw exceptions whose classes are subclasses of `UncheckedException`. For example, when compiling the command

```
     x = v[i];
```

the compiler will generate code to test if `i` is within the legal array bounds. This test will belong to the program run-time system and may throw exception `IllegalArrayIndexException`. All the exceptions the run-time system may throw belong to subclasses of `UncheckedException` and need not to be declared as part of the type of the `exception` method parameter. However, if the programmer wants to throw this exception herself, she must declare it:

```
proc aMethod()
            ( exception : CatchIllegalArrayIndexException )
```

```
class Catch
  proc init()  begin  initialize();  end
  public:

    proc initialize()
      begin   exceptionObject = nil;   classException = nil;
      wasFixed = false;
      end
    proc set( p_exceptionObject : Exception;
              p_classException : AnyClassObject )
      begin
      exceptionObject = p_exceptionObject;
      classException = p_classException;
      end
    proc getException() : Exception
        // returns the exception object thrown by exception.throw(obj)
      begin
      return exceptionObject;
      end
    proc getClassException() : AnyClassObject
        // returns the class of the object exception
      begin
      return classException;
      end
    proc wasThrown() :  boolean
      begin
      return exceptionObject <> nil;
      end
    proc fixed() : boolean
      begin
      if wasThrown() and wasFixed  then wasFixed = false;  return true;
      else return false;  endif
      end
    proc setFixed( p_wasFixed : boolean )
      begin
      wasFixed = p_wasFixed;
      end
  private :
    var exceptionObject : Exception;  classException :  AnyClassObject;
        wasFixed : boolean;
end
```

Figura 11.1: Class `Catch`

```
  begin
  ...

  exception.throw( IllegalArrayIndexException.new() );
  ..
  end
```

The program can catch exceptions of subclasses of `UncheckedException` even when they are not declared, which is the normal case. Example:

```
class FigureSet
  public:
    proc add( f : Figure )
      /* add a figure to the set. Since there may not be sufficient
         memory, exception OutOfMemoryException (not declared) may be
         thrown. */
    ...
end


...
var x : Square;
var fig : FigureSet;
...

try( CatchOutOfMemory.new() )
  fig.add(x);
end
```

In the previous example, the program need not to use the `try-end` statement since `add` may only throw unchecked exceptions.

Class object `CatchAll` is subtype of `Catch` and has a method
  `proc throw( exc : Exception )`
Then `CatchAll` can be used to catch all exceptions:

```
try(CatchAll)
  f = File.new("article.doc");
  ...
end
if CatchAll.wasThrown()
then
  Out.writeln("There was an error ...");
endif
```

Method `wasThrown` returns `true` if an exception was thrown inside the `try-end` command.

Class object `HCatchAll` is similar to `CatchAll` but it finishes the program when an exception is thrown. A message is printed using object `Out`.

The execution of a program begins in a compiler-created object called `MainProgram` that catchs all unchecked exceptions that may be raised by the user program. This object is defined as

```
object MainProgram
```

```
  public:
    proc run( theArgs : String )
      begin
      var args : array(String)[];
        // parse theArgs resulting in args
      ...
      var catchUnchecked : HCatchUncheckedException;
      catchUnchecked#init();
      try(catchUnchecked)
        UserClass.run(args);
      end
      end
end
```

UserClass is the class object in which the program must begin its execution.

The type of object `exception` declared in a method `m` must have `throw` methods with exceptions the `m` method can throw. Then, if method

```
  proc m()
        (exception : CatchX)
    begin
    ...
    end
```

can throw exceptions `FileOpenException` and `ParseException`, class `CatchX` must have at least methods

```
     proc throw( exc : FileOpenException )
     proc throw( exc : ParseException )
```

The class

```
class CatchFruitException subclassOf CatchUncheckedException
  public:
    proc init()
      begin
      end
    proc throw( exc : FruitException )
      begin
      ...
      end
    proc throw( exc : BananaException )
      begin
      ...
      end
end
```

defines two methods `throw` in which one of the parameter types (`BananaException`) is subtype of the other (`FruitException`). This introduces a kind of redundancy in the class since a `BananaException` object can be passed as a parameter not only to method "`proc throw(exc : BananaException`" but also to "`proc throw(exc : FruitException)`". Let us see an example explaining how the compiler manages this.

In a code like

```
catchFruitException = CatchFruitException.new();
try(catchFruitException)
  makeJuice();
end
```

in which `makeJuice` executes the message send

```
exception.throw( BananaException.new() )
```

the run-time system will look for a `throw` method with a parameter type that is supertype of `BananaException`. This search is made in the stack of objects of `try` statements. First the run-time system pops the first object off the stack and starts a linear search in the object class for a method

```
proc throw( exc : E )
```

such that `E` is supertype of the poped object class.[2] This search is made in the order given by the declarations of methods in the text of the catch class. If no `throw` method of this first object is adequate, the next object is poped off the stack of `try` objects and the search continues.

This is the only Green feature in which the order of method declaration does matter. Notice that catch objects can be class objects — the only requirement for one catch object is that it must be subtype of predefined class `CatchUncheckedException`.

Suppose class `CatchA` declares a method

```
throw( exc : BananaException )
```

and class `CatchB`, which inherits from `CatchA`, defines a method

```
throw( exc : FruitException )
```

If an object of `CatchB` is used in a try block, the search for a handler is made first in `CatchB` and then in `CatchA`. So, an exception signalling

```
try(aCatchB)
  exception.throw( BananaException.new() );
end
```

will cause a search in `CatchB` and method `throw(FruitException)` will be used even though method `throw(BananaException)` of the superclass would be more specific.

In class `CatchFruitException`, method `throw` with a `FruitException` parameter appears before method with a parameter type `BananaException`. Therefore the call

```
exception.throw( BananaException.new() )
```

in `makeJuice()` will execute method

```
CatchFruitException::throw( exc : FruitException )
```

Since the correct thing should be to call

```
CatchFruitException::throw( exc : BananaException )
```

this method should be put before `throw(exc : FruitException)` in its class declaration.

Every catch class or catch class object should be subtype of class `CatchUncheckedException`. A catch object will keep a reference to the class of the exception object that was thrown. This reference can be got by method `getClassException`. Using this feature our fruit example could be rewritten as

```
catch = CatchFruitException.new();
try(catch)
  makeJuice();
```

[2]If this object is a class object, the test is to discover if `E` is supertype of the class object type.

```
end
case catch.getClassException() of
  FruitException :
    Out.writeln("Fruit rotted");
  BananaException :
    Out.writeln("Cannot make a juice of bananas");
end
```

We supposed in this example that the `throw` methods of `CatchFruitException` had empty bodies. The real exception treatment is made in the case statement after `try-end`. If `makeJuice` raises an exception `OrangeException`, subclass of `FruitException`, method

```
proc throw( exc : FruitException )
```

of object `catch` will be executed since the catch class, `CatchFruitException`, does not have a method

```
proc throw( exc : OrangeException )
```

and `OrangeException` is a subtype of `FruitException`. The the call

```
catch.getClassException()
```

of the case statement will in fact return class `FruitException` instead of `OrangeException`. However,

```
catch.getException()
```

would return the `OrangeException` object thrown by `makeJuice` and

```
catch.getException().getClassObject()
```

would return class `OrangeException`.

The above syntax mimics the exception system of Java and C++. This example can be written in Java as

```
try {
  makeJuice();
}
catch ( BananaException exc )
{
  System.Out.writeln("Fruit rotted");
}
catch ( FruitException exc )
{
  System.Out.writeln("Cannot make a juice of bananas");
}
```

Note the order of the catch clauses is important here. The exception object can be accessed in this Java code using parameter `exc` of the catch clauses. In the Green code the exception object has to be retrieved by the call "`catch.getException()`".

In the previous Java example, the compiler will issue an error message if the `try` block may throw an exception not presenting in the catch clauses and that was not declared in the method header. We hope the Green compiler will also issue an error in the same situation when simulating the `catch` clauses of Java with the case statement as shown in the last Green example. That is, the compiler should issue a warning if `CatchJuiceException` declared method

```
proc throw( exc : ReadFileException )
```

since there is no correspondent `case` label to treat this case.

We hope every class object of a catch class will have a `get` method for returning an object of the catch class. Then only one object of the catch class need to be allocated in the whole program. The following code illustrates this mechanism.

```
object CatchFileOpenException
  public:
    proc init()
      begin
      aCatch = CatchFileOpenException.new();
      end

    proc get() : CatchFileOpenException
      begin
      aCatch.initialize();
      return aCatch;
      end
  private:
    var aCatch : CatchFileOpenException;
end

class CatchFileOpenException subclassOf CatchUncheckedException
  public:
    proc init()
      begin
      end

    proc throw( exc : FileOpenException )
      begin
      ...
      end
end
```

11.4 Fixing Errors with Catch Objects

After an exception is thrown, it will be catch by a catch object that appears in a `try-end` command. A method `throw` of the catch object is called passing the exception object as parameter. The `throw` method may try to fix the error using information from the exception object. To illustrate that, we will use the following classes

```
class WriteErrorException subclassOf Exception
  public:
    proc init( pf : File )
      begin
      f = pf;
      end
    proc getFile() : File
      begin
      return f;
      end
```

```
 private:
   var f : File;
end



class CatchWriteErrorException subclassOf CatchUncheckedException
  public:
    proc init()
      begin
      end
    proc throw( exc : WriteErrorException )
        var success : boolean;
            f       : File;
      begin
      f = exc.getFile();

        /* tries to fix the error by changing the file name, writing to
           another drive or directory, changing the attributes of the
           file. That means putting the state of the object exc in a
           state that will not produce this error anymore */
      ...
        // if successful, set fixed to true
      setFixed(success);
      end
end



class File
  public:
    ...
    proc save()
             ( exception : CatchWriteErrorException )
      begin
      ...
      if error
      then
        exception.throw( WriteErrorException.new(self) );
      endif
      end
  private:
    ...
end


class Test
  public:
```

```
    ...
    proc saveAll()
      begin
      ...
      catch = CatchWriteErrorException.new();
      repeat
        try(catch)
          f.save();
          ...
        end
      until catch.fixed();
      end  // saveAll
end
```

Method `saveAll` saves files to disk through statement "`f.save()`" which may throw an exception `WriteErrorException`. The thrown exception keeps a reference to the file that could not be saved to disk — see `File::save`. If the exception is throw in the above example, method

`CatchWriteErrorException::throw`

will be executed. This method tries to fix the error by asking the user directions on what to do. Method `throw` may:

- change the file name. The original name may be illegal;
- write the file to another drive/directory;
- change the file attributes. The file may be read only.

In general, a method `throw` should try to put the object (`f` in this case) in an error-free state. The object whose defective state caused the exception should be passed as a parameter to the exception object. Then method `throw` can access this object through the exception object.

After `CatchWriteErrorException::throw` is executed, the execution continues after the `try-end` command. Then "`catch.fixed()`" is executed. Method `fixed` returns true when an exception was thrown and the error was fixed thus enabling the `try-end` command to be executed again. The `throw` method of the catch object that fixes the error should call method `setFixed` of the catch object.

11.5 Comparison with Other Exception Systems

By comparing the example in Java and Green we conclude the Green exception system is at least equivalent to the Java system. In fact, the Green system is much more. It casts catch clauses in `throw` methods of `Catch` classes making them reusable. Green shapes exceptions in a more object-oriented fashion by grouping what would be Java catch clauses in Green `Catch` classes. Not only catch classes can be reused but also catch objects that can be passed as parameters, stored in variables, and so on. One can imagine a factory class [3] which returns catch objects appropriate to the current environment. For example, in a command-line environment the factory class would return catch objects that would print error messages in the standard output. If a graphical environment is used, the messages could be shown in windows and so on.

Catch classes can be subclassed and `throw` methods overridden thus providing code reuse of error treatment.

Java requires a method to declare the exceptions it may throw[3] through a special syntax:

```
void makeJuice()
     throws FruitException, BananaException
{
   ...
   throw new FruitException();
}
```

Green requires the declaration of exceptions in an object-oriented way:

```
proc  makeJuice()
               ( exception : CatchFruitException )
begin
...
exception.throw( FruitException.new() );
...
end
```

`exception` is an implicit parameter of `makeJuice`. This means the thrown of an exception follows the rules of message sends. And, since one puts a class like `CatchFruitException` as the parameter type, it is not necessary to list all exception classes in the method header. This may be cumbersome to do if a lot of different kinds of exceptions can be thrown by a lot of methods, although in general the number of exception classes a method uses is small.

11.6 Exception and Meta-Level Programming

The `exception` parameter in Green is a meta-level parameter automatically managed by the compiler. We can imagine the program being executed in a plane of a three-dimensional space called "application plane". There is a "meta-level exception" plane parallel to the application plane. The execution in both planes are synchronized. While at the application plane there is a stack of called methods, in the meta-level exception plane there is a stack of catch objects. The application plane manages parameter passing to methods just as ordered by the program. These parameters correspond to the first parameter list after the method name, as `elem` in

```
proc search( elem : integer )    // first parameter list
           ( exception : CatchNotFoundException ) : integer  // second list
  var i : integer;
begin
...
  // didn't find anything
exception.throw( NotFoundException.new( elem, i ) );
...
end
```

The second parameter list,

`( exception : CatchNotFoundException )`

corresponds to the meta level exception plane which is automatically managed by the compiler.

[3]Some exceptions in Java need not to be declared — they are unchecked.

The `exception` parameter can only be used as a message receiver. It cannot be used as a message-send real parameter or in the left or right-hand side of an assignment. This restriction could be lifted if we considered `exception` has class `CatchException` that inherits from `Catch` and defines the method

```
    proc throw( exc : Exception )
```

However, this would make things confusing because the declared type of `exception` may be different (and generally is) from `CatchException`. The `exception` type would be the declared type in a message send and `CatchException` anywhere else.

Whenever the program execution reaches a `try(catch)` statement, the catch object is pushed into the stack of catch objects. After the `try` statement finishes its execution, the object is removed from the stack.

If a method does not declare an exception parameter, the compiler introduces one whose type is `CatchUncheckedException`. Then a method

```
    proc set( i : integer )
```

in fact is

```
    proc set( i : integer )
            ( exception : CatchUncheckedException )
```

Through this `exception` parameter the compiler can throw exceptions like `OutOfMemoryException` or `MessageSendToNilException`, which are unchecked. Since programmers normally do not throw unchecked exceptions, they do not declare an exception parameter in methods that can only throw this kind of exceptions.

Object `Runtime`, discussed in Chapter 13 and Appendix B, allows one to inspect the catch object stack. At run time the program can discover what exceptions can be caught and it can even attach a shell to a catch object of the catch object stack. See Chapter 12 for definition of shells.

11.7 Static Type Correctness of Exceptions

Class `CatchFruitException` can catch exceptions of classes `FruitException` and `BananaException`. Assume a class `CatchBurnedFoodException` can catch exceptions of class `BurnedFoodException` and method `heat` can `throw` exceptions of classes `FruitException`, `BananaException`, and `BurnedFoodException`. Then in a method

```
proc prepareFood( f : Food )
                ( exception : CatchFruitException )
    var catch : CatchBurnedFoodException;
  begin
  catch = CatchBurnedFoodException.new();
  try(catch)
    heat(f);
    split(f);
  end
  end
```

There will not be any type error relating to exceptions. Any exception raise by `heat` will be catch by the `try` statement or by the method that called `prepareFood`.[4]

That means the type of `exception` inside the `try-end` statement is enhanced by the type of `CatchBurnedFoodException`. Inside this `try-end`, the type of `exception` is:

[4]Or in fact another method that indirectly called `prepareFood`.

```
proc throw( exc : FruitException )        // from the exception object
proc throw( exc : BananaException )       // from the exception object
proc throw( exc : BurnedFoodException )   // from the catch object
```

What happened is that the `exception` object passed as a parameter to `prepareFood` was the top element of the stack of catch objects. The `try` statement pushed object `catch` into this stack enhancing the type of `exception`. This occur because inside the `try-end` statement, a message sent to `exception` is searched first in the object of the top of the catch stack. If the method is not found there the search continues in the stack from top to bottom making the run-time type of `exception` to be the union of the types of all objects in the catch stack. However, the compile-time type is more restricted because an `exception` parameter assumes the type it is declared with, at least outside any `try-end` statement.

11.8 Exceptions and Enhancing of Objects at Runtime

The casting of catch clauses in an object-oriented form reveals an interesting aspect of exceptions: an `exception` object has a dynamic type which depends on the types of objects in an object stack. Since the object stack grows and shrinks at run time, so does the `exception` type. When the program execution reaches the `try` command, a catch object is pushed into the stack thus adding method signatures to the `exception` type. When the execution of a `try-end` command ends, the catch object is poped off the stack thus making the `exception` type return to its previous value. This dynamic type modification would cause run-time type errors if it were not restricted to a block defined at compile time, which is the `try-end` block.

Exceptions cannot be cast in a pure object-oriented form for two reasons:

- the dynamic nature of the `exception` type;
- the urgence an exception should be treated: the first catch object that can handle the thrown exception object is used.

This last item will become clear after we try to solve the former item by using shells (Green metaobjects — see Chapter 12). Suppose Green supports a modified kind of shells in which a shell class

```
shell class Border(Window)
  public:
    proc init( c : integer )
      // methods not defined in Window
    proc setBorderColor( c : integer )
      ...
    proc getBorderColor() : integer
    ...
end
```

can declare in its public section methods not defined in class `Window`. When a shell of `Border` is attached to a `Window` object, the object can be cast to class `BorderedWindow`, which is a subclass of `Window` and defines the methods of shell class `Border`. See the example.

```
var window : Window;
```

```
var borderedWindow : BorderedWindow;
...
 Meta.attachShell( window, Border.new( Color.blue ) );

borderedWindow = BorderedWindow.cast(window);
Out.writeln( borderedWindow.getBorderColor() );
...
```

`BorderedWindow` has no inheritance relationship with shell class `Border`.

The message send "`borderedWindow.getBorderColor()`" will call method `getBorderColor` of shell class `Border`: `borderedWindow` points to a `Window` object whose type was enhanced by the shell attachment. This dynamic type modification can easily result in type errors:

```
Meta.removeShell(window);
borderedWindow.setBorderColor(Color.green);
```

The last statement will result in a run-time type error since the object pointed by `window` and `borderedWindow` does not have a `setBorderColor` anymore.

This kind of error does not occur if only one variable refers to the object and its type is enhanced only in a specific region of the code defined at compile time such as the block of a `try-end` command. Now we are going to introduce a new syntax in which `exception` objects will have their types enhanced by catch classes. This syntax is not legal in Green — it is only used to exemplify the ideas. The code

```
proc prepare( f : Food )
            ( exception : CatchReadFileException )
  var catch : CatchBananaException;
begin
catch = CatchBananaException.new();
try(catch)
  prepareFood(food);
end
end
```

is written as

```
proc prepare( f : Food )
            ( exception : CatchReadFileException )
  begin
  try(CatchBananaException.new() )
    prepareFood(food);
  end
  end
```

and translated by the compiler into

```
proc prepare( f : Food )
            ( exception : CatchReadFileException )
  begin
  Meta.attachShell( exception, CatchBananaException_ReadFileException.new() );
    // new scope --- illegal in Green but legal in this example
    begin
```

```
    var exception : CatchBananaException_ReadFileException; // 1
    exception = CatchBananaException_ReadFileException.cast( prepare::exception );
    prepareFood(food);
    end
  end // prepare
```

There are some observations to be made here. First, `CatchBananaException_ReadFileException` is the compiler created class

```
shell class CatchBananaException_ReadFileException(CatchReadFileException)
  public:
     // all the methods of CatchBananaException
end
```

Shells of this class can be attached to `CatchReadFileException` objects as parameter `exception`. The result is an object that has methods of classes `CatchBananaException` and `CatchReadFileException`. See the "`attachShell`" message to `Meta` in `prepare`. But that does not mean a message send

```
     exception.throw( BananaException.new() )
```

is legal since the `exception` type is `CatchReadFileException` which does not have the method

```
     proc throw( exc : BananaException )
```

It is necessary to *cast* `exception` into a type that does have this method, which is made in the line following line `// 1`.

We are assuming the `begin-end` block inside the `begin-end` block of method `prepare` introduces a new scope, a rule not valid in Green. The `exception` variable declared in this block (line `// 1`) is initiated with the `exception` variable of the outer scope. That is, "`prepare::exception`" means the parameter `exception` of `prepare`. Again, this is not valid Green syntax.

`prepareFood` may have been declared as

```
proc prepareFood( f : Food )
                ( exception : CatchBananaReadException )
  begin
  ...
  exception.throw( BananaException.new() );
  ...
  end
```

in which `CatchBananaReadException` has all methods of `CatchBananaException` and `CatchReadFileException`.

The call

```
     prepareFood(food);
```

in method `prepare` is legal because it is translated to

```
     prepareFood(food)
                (exception) // passes exception as a meta-level parameter
```

and `exception` has the same type as `CatchBananaReadException`. This last statement is true because of the cast

```
     exception = CatchBananaException_ReadFileException.cast(prepare::exception );
```

Note this cast is not legal in Green because shell classes do not exist at run time (they are not real classes) and therefore they do not support method `cast`.

The statement

```
     exception.throw( BananaException.new() );
```

of `prepareFood` would call method

```
proc throw( exc : BananaException )
```

of object `exception` since this is a normal message send.

Now suppose `prepare` was modified to

```
proc prepare( f : Food )
            ( exception : CatchReadFileException )
  begin
  try( CatchBananaException.new() )
    try( CatchFruitException.new() )
      prepareFood(f);
    end
  end
  end
```

Using the implementation of exceptions just described, based in shell attachements, the statement

```
exception.throw( BananaException.new() )
```

in `prepareFood` continues to call the same method of object `exception` as before,

```
proc throw( exc : BananaException )
```

which means the inner `try-end` command of `prepare` was skipped and the control transferred to the outer `try-end`. That is not the best alternative since the inner `try-end` command can also treat the exception and fewer `try-end` commands should be poped off the stack. In general one wants to treat an error as soon as possible to avoid the premature ending of methods in the call stack.

As we have seen, this does not happen if shells are used to implement exceptions because the following situation may occur:

- the topmost `try` can treat exception `BananaException`;
- a `BananaException` is thrown and treated by the bottom-most `try` command thus making a great number of methods finish their execution.

That justifies the claim we have make in the beginning of this section: exceptions cannot be cast in a pure object-oriented form because an exception should be treated as soon as possible. when we cast exceptions in a pure object-oriented form, all `throw` methods are put in a single class losing therefore the relative order of the `try` statements that furnishes these methods. Then a `throw` method may be invoked when another nearest could be called. Note we assumed a message send

```
exception.throw(exc)
```

causes a search at run time for the most specific method that accepts parameter `exc`. This is not valid for overloaded methods in Green. Note that although we have used shells to prove exceptions cannot be made completely object-oriented, there would be no difference if we have used any other mechanism to enhance the type of an object. For example, we could have created a class `CatchBananaReadTwoException` in which some `throw` methods were forwarded to an instance variable:

```
class CatchBananaReadTwoException subclassOf CatchReadFileException
  public:
    proc init( f : CatchReadFileException )
      begin
```

```
      self.f = f;
      end
    proc throw( exc : ReadFileException )
      begin
      f.throw(exc);
      end
    proc throw( exc : BananaException )
      begin
        // treats the exception
      ...
      end
  private:
    var f : CatchReadFileException;
end
```

Method `prepare` would become:

```
proc prepare( f : Food )
            ( exception : CatchReadFileException )
    var exception : CatchReadFileTwoException;
  begin
  exception#init(prepare::exception);
  prepareFood(food);
  end
```

11.9 Assertions and Exceptions

When an assertion of any method fails, an exception of class `AssertionBeforeException` or `AssertionAfterException` is thrown. Each of these classes has a constructor

```
proc init( mi : MethodInfo )
```

`MethodInfo` is a class of the introspective class library that gives information about the method such as its name, class, and so on.

11.10 Subtyping and Exceptions

It is time to define subtyping rules that include exceptions. The type of a class is the set of its method signatures as before. But now a method signature includes the type of parameter `exception`. Therefore, the signature of method

```
proc get( i : integer )
        ( exception : CatchIllegalIndexException ) : Any
```

is

```
get(integer)(CatchIllegalIndexException) : Any
```

A type $S = \{\mathtt{n}_1,\ \mathtt{n}_2,\ \ldots\mathtt{n}_p\}$ is equal to type $T = \{\mathtt{m}_1,\ \mathtt{m}_2,\ \ldots\mathtt{m}_q\}$ ($S = T$) if $p = q$ and $\mathtt{n}_i = \mathtt{m}_i$, $1 \le i \le p$. The relation $=$ for methods is defined as follows.

Let

$\mathtt{n}(T'_1,\ T'_2,\ \ldots T'_k)(E')\ :\ U'_1$

$\mathtt{m}(T_1,\ T_2,\ \ldots T_t)(E)\ :\ U_1$

be the signatures of two methods. We say that $\mathtt{n} = \mathtt{m}$ if

- `m` and `n` have the same name.
- $t = k$.
- $T_i = T'_i$, $1 \leq i \leq t$.
- $U' = U$.
- $E' = E$.

The definition of subtype remains the same. That is, S is subtype of T is $T \subset S$.

Since a subclass is a subtype, a superclass method overridden in a subclass should keep its signature which includes keeping the type of parameter `exception`. This could be different: it could be legal in Green to declare the type of `exception` in an overridden subclass method to be a supertype of the type of `exception` in the superclass method. That is, if

```
proc get(integer)(CatchJuiceException) : Any
```

is the superclass method it could be legal in Green to override this method to

```
proc get(integer)(CatchFruitException) : Any
```

in a subclass in which `CatchFruitException` is a supertype of `CatchJuiceException`. The sole reason this is illegal in Green is that it would make the language more complex. There is no logical error in this feature as explained in the following paragraphs.

Suppose class `JuiceMachine` defines a method

```
proc makeJuice()
                ( exception : CatchJuiceException )
```

which is overridden in a subclass `SpecialJuiceMachine` as

```
proc makeJuice()
                ( exception : CatchFruitException )
```

Suppose `CatchJuiceException` has methods

```
proc throw( exc : FruitException )
proc throw( exc : DryException )
```

If `CatchFruitException` has method

```
proc throw( exc : FruitException )
```

then the redefinition of `makeJuice` interface in `SpecialJuiceMachine` would be correct because `CatchFruitException` is supertype of `CatchJuiceException`. No run-time type error can occur because of this redefinition of `makeJuice` with a different interface. To understand that, let us study the following code

```
catch = CatchJuiceException.new();
try(catch)
  var specialJuiceMachine : SpecialJuiceMachine;
    // ok, assignment of the kind "Type = subtype"
  juiceMachine = SpecialJuiceMachine.new();
  juiceMachine.makeJuice();
end
```

The method `makeJuice` called in this code will be

```
SpecialJuiceMachine::makeJuice()
```

The exceptions this method may throw will be catch by the enclosing `try-end` command because:

- method `SpecialJuiceMachine::makeJuice` can throw a subset of exceptions that `JuiceMachine::makeJuice` can since `CatchFruitException`, type of `exception` in

`SpecialJuiceMachine::makeJuice`, has less `throw` methods than `CatchJuiceException`, the type of `exception` in `JuiceMachine::makeJuice`. That is, `CatchFruitException` is supertype of `CatchJuiceException`. As stated before, the language requires that from overridden subclass methods;

- the `try-end` command uses a `catch` object appropriate to handle exceptions of `JuiceMachine::makeJuice` since there is a

  statement

  ```
  juiceMachine.makeJuice()
  ```

  in which the declared type of `juiceMachine` is `JuiceMachine`. Then the `try` command in fact will catch exceptions of less types than it is able to catch. Then there cannot be any run-time error.

Capítulo 12

Shells and Dynamic Extensions: Metaprogramming in Green

This Chapter describes dynamic extensions and the Green metaobjects called shells. A metaobject controls the behavior of a single object. Any message sent to the object is packed into an object `Message` and delivered to the object metaobject. The metaobject can then call the corresponding object method, forward the message to another object, and so on. A dynamic extension can be attached to a class at run time. It will replace the class methods by the corresponding methods of the extension class.

Shells and dynamic extensions are discussed deeply by Guimar aes [4]. In this report we will define these features without further discussing them. The definitions given here are slightly different and more generic than that of [4].

12.1 Shells

A shell is a pseudo-object with methods and instance variables that can be attached to an object as graphically represented in Figure 12.1. Any message sent to the object will be first searched in the shell and then in the object.

A shell class `Border` is defined in Figure 12.2. A shell of class `Border` can be attached to objects of `Window` and its subtypes as specified by the line

```
shell class Border(Window)
```

Class `Window` is shown in Figure 12.3. There may not be any class object associate to a shell class. If this were allowed, shells would be more complex since we could have a class object without an associate class (shell classes are not classes !).

The set of public method interfaces of `Border` must be a subset of the set of method interfaces of `Window`. If we consider a shell class has a type, then the type of `Border` should be supertype of `Window`.

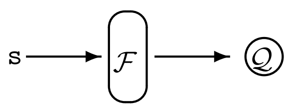

Figura 12.1: A shell $\mathcal{F}$ attached to an object $\mathcal{Q}$

```
shell class Border(Window)
  proc init()
    begin
    end
  public:
    proc draw()
      begin
      self.drawBorder();
        // call object method
      super.draw();
      end
    private:
      proc drawBorder()
        ...
end
```

Figura 12.2: A shell class to subtype-`Window` objects

```
class Window
  proc init()
    begin
    ...
    end
  public:
    proc draw()
      ...

    ...
end
```

Figura 12.3: Class `Window` with at least method `draw`

```
object Meta
  public:
      // only the method interfaces are shown
    proc attachShell( any : IdentAST; exp : ExprAST )
    proc removeShell( any : IdentAST )
    proc attachExtension( aClass : ClassNo; dynExt : ExtensionClassNo )
    proc removeExtension( aClass : ClassNo )
end
```

Figura 12.4: Interface of the `Meta` module

A shell of `Border` can be attached to a `Window` object `window` through the syntax

    Meta.attachShell(window, Border.new())

`Meta` is a module and modules are compile-time objects. The `Meta` module interface is shown in Figure 12.4. `IdentAST`, `ExprAST`, `ClassNo`, and `ExtensionClassNo` are the classes of the compiler Abstract Syntax Tree (AST) that represent an identifier, an expression, a class, and a dynamic extension class, respectively.

The message send of `attachShell` to `Meta` at compile time will generate a run-time message send that will create a shell `Border` and attach it to object `window`.

Class-`Border` methods can call methods of the object the shell is attached to by sending messages to `super`. As an example, let us study the code

```
window = Window.new("My Window", 30, 20, 120, 300);
window.draw(); // call Window::draw()
Meta.attachShell( window, Border.new() );
window.draw();
```

The last statement will call `Border::draw` that will draw a border through its first statement "`self.drawBorder()`" and execute "`super.draw()`", which will call `Window::draw`.

This is the same as if `Border` inherited class `Window`. We can attach a `Border` shell to an object of a `Window` subtype:

```
var window : Window;

  // ColorWindow is subtype (maybe subclass) of Window
window = ColorWindow.new("My Window", Color.green, 30, 20, 120, 300);
  // call ColorWindow::draw()
window.draw();
Meta.attachShell( window, Border.new() );
window.draw();
```

The last message send will execute `Border::draw` that will call `Border::drawBorder` ("`self.drawBorder()`") and `ColorWindow::draw` ("`super.draw()`").

A shell of class `Border` can be attached to objects of `Window` and its subtypes which may include classes that are not subclasses of `Window`. Therefore the calls to `super` in `Border` methods can only refer to public `Window` methods.

Shells can be removed by method `removeShell`:

    Meta.removeShell(window)

This command removes the last shell attached to object `window`. There may be more than one shell attached to an object. Shells are attached and removed in a stack fashion.

If there is no shell attached to `window`, this message send throws exception `NoShellException`.

The above code may throw the following exceptions:

    OutOfMemoryException
    ClassNotInAllowedSetException

These exceptions can be caught by the class `CatchMetaException` that also catches

    NoShellException
    NoExtensionException

This last class is discussed ahead. In fact, these exception classes compose a hierarchy:

```
MetaException
  ClassNotInAllowedSetException
```

```
NoShellException
NoExtensionException
```

The `throw` methods of `CatchMetaException` do nothing — they do not finish the program or print any error message. There is also a class `HCatchMetaException` whose methods print an error message and finishes the program.

It is interesting to note that, in a statement

```
Meta.attachShell( obj, ShellClass.new() )
```

no compile time error can be issued based on the declared type of `obj` and on `ShellClass` type. The declared type of `obj`, say, `ObjClass`, may have a subtype that has all methods of `ShellClass`. This is necessary, although not sufficient, to make the statement legal at run time.

12.2 A High-Level View of Shell Implementation

The examples of the previous section showed attachements of `Border` shells to objects of classes `Window` and `ColorWindow`. These classes belong to the "allowed set" of `Border`, the set of all classes to whose objects `Border` shells may be attached at run time. If a shell `Border` may be attached to a `NiceWindow` object at run time, the programmer should specify this fact through a compiler option. Of course, no class of the allowed set can be abstract. For each class `A` of the `Border` allowed set, the compiler creates a special class `Border$s$A` that inherits from `A` and has all class-`Border` methods, including method bodies.

The statement

```
Meta.attachShell( window, Border.new() )
```

is a compile time call to object `Meta`. This needs to be a compile time message because it contains an expression "`Border.new()`" which is illegal in Green: shell classes like `Border` are not classes and cannot receive the `new` message. In fact the compiler will get around this problem by executing method `init()` of `Border`[1] on object `window` ("`window.init()`") after the shell was attached to it. That is, the compile-time message send

```
Meta.attachShell( window, Border.new() )
```

will be replaced by the statements

- *find the class of object* `window` *which will be called* `C`*;*
- *check whether* `C` *is in the allowed set of* `Border`*. If not, throw exception* `ClassNotInAllowedSetException`*;*
- *there is a class* `Border$s$C`*. Then change the class of* `window` *to* `Border$s$C`*;*
- *create an object* `obj` *with all instance variables of* `Border`*. If there is not enough memory, throw exception* `OutOfMemoryException`*. Insert the pair (*`window`*,* `obj`*) in a hash table. Later on,* `obj` *(with the instance variables) will be retrieved using* `window` *as key to the table;*
- *calls method* `Border::init` *on* `window`*. This is as if there were a message send*

  ```
  window.Border::init()
  ```

  in which the method to be executed is fixed at compile time — `init` *of* `Border`*.*

The compiler needs the source code of the shell class in order to create a new class like `Border$s$Window`. Shell classes are much like C++ *templates* in which the parameter is the superclass; that is, a class of the `Border` allowed set.

[1] Method `new()` of class object `Border` corresponds to class-`Border` method `init`.

12.3 Shell Inheritance

Shell classes can inherit from other shell classes. If a shell class `C` inherits shell class `B` and class `A` belongs to the allowed set of `C`[2], the compiler will create a class `C_B_A` such that this class:

- has the source code of `C`;
- inherits from class `B_A`.

Class `B_A` :

- has the source code of `B`;
- inherits from `A`.

Then any message sends to `super` inside class `C` will cause a search for a method in `B` and then in `A`.

Maybe shell classes will be able to inherit from normal classes. This opens the possibility of any class be treated as a shell class. This can be seen in the example

```
shell class Empty(Window) subclassOf ColorWindow
end
```

in which `ColorWindow` is a subclass of `Window`. A shell class `Empty` has everything `ColorWindow` has thus effectively treating `ColorWindow` as a shell class. Note the compiler would need the source code of `ColorWindow` and all of its superclasses for the same reasons it needs the source code of the shell classes.

12.4 Shell Initialization

A shell class can define a method `init` to initiate the shell object.[3] This may be used to communication between the shell and some external control of the object. For example, the shell

```
shell class DrawCount(Window)
  proc init( counter : Counter )
    begin
    self.counter = counter;
    end
  public:
    proc draw()
      begin
      counter.add(1);
      super.draw();
      end
  private:
    var counter : Counter;
end
```

can be used with the class

[2]That means shells of `C` can be attached to `A` objects.

[3]Although shells are not exactly objects, we will treat them as such.

```
class Counter
  proc init()
    begin
    n = 0;
    end
  public:
    proc add( s : integer )
      begin
      n = n + s;
      end
    proc get() : integer
      begin
      return n;
      end
  private:
    var n : integer;
end
```

to count the number of times the `draw` method of a `Window` object was called:

```
var catch : CatchMetaException;
var c : Counter;
var w : Window;

w = Window.new(...);
c = Counter.new();
try(catch)
  Meta.attachShell( w, DrawCount.new(c) );
end
...
Out.writeln("draw was called ", c.get(), " times");
```

Since shells are not really objects, they cannot be referenced by any variable. Therefore there is no way of getting the value of an instance variable of a shell object. So, `DrawCount` could not have been implemented as

```
shell class DrawCount(Window)
  proc init()
    begin
    n = 0;
    end
  public:
    proc draw()
      begin
      n = n + 1;
      super.draw();
      end
  private:
    var n : integer;
```

```
end
```

No one outside `DrawCount` will ever access the value of `n`.

12.5 Reflective Shell Classes

A class can be declared as

```
reflective class Window
   ...
end
```

This means that, if a shell is attached to a `Window` object, the access to shell instance variables will be faster than if `Window` were declared as a normal class. All subclasses of `Window` must be declared as "`reflective`" as well.

12.6 Shells and Class Objects

A shell may be attached to an object that is a class object. An example is shown in Figure 12.5 in which a `PlanetManager` class controls the number of planets created from class `Planet`. Note:

- there is no need to declare a class object `Planet` since the compiler will do this;
- class object `Planet` has methods

  ```
  proc new( name : String; radius, distanceSun : real )
  proc shallowClone() : Any
  proc deepClone() : Any
  ...
  ```

  These method interfaces compose the type of class object `Planet` which is represented by `type(Planet)`.

A shell is attached to `Planet` by

```
Meta.attachShell( Planet, PlanetManager.new() )
```

After this, the command

```
saturn = Planet.new("Saturn", SaturnRadius, SaturnDistanceSun);
```

will call `PlanetManager::new()` that will then call method `new` of class object `Planet` in the statement

```
return super.new(name, radius, distanceSun);
```

Note `PlanetManager` can be used even if the source code of `Planet` is not available.

12.7 Method `interceptAll`

One can declare a method

```
proc interceptAll( mi : ObjectMethodInfo;
                   vetArg : array(Any)[] )
```

in a shell class to intercept all messages sent to the object the shell is attached. `ObjectMethodInfo` is a class of the Green Introspective Class Library. An object of this class describes a specific method of an object. `vetArg` is an array containing the real parameters used to call method `mi`.

```
class Planet
  proc init( name : String; radius, distanceSun : real )
    begin
    ...
    end

  public:
    ...
end


shell class PlanetManager( type(Planet) )
  proc init()
    begin
    numberOfPlanets = 0;
    end

  public:

    proc new( name : String; radius, distanceSun : real ) : Planet
      begin
      if numberOfPlanets >= 9
      then
        Out.writeln("Unknow solar system: too many planets");
          // ends the program
        Runtime.exit(1);
      endif
      ++numberOfPlanets;
      return super.new(name, radius, distanceSun);
      end

  private:
    var numberOfPlanets : integer;
end
```

Figura 12.5: A planet manager that controls the number of planets

As usual, basic class values are wrapped in classes `Char` and the like before they are inserted in `vetArg`.

When a message `m` is sent to an object $\mathcal{Q}$ with a shell, method `m` of the shell will be executed, if one exists. If the shell does not have a method `m` but does have an `interceptAll` method, the message parameters are packed in an array used as the argument `vetArg` in a call to method `interceptAll` of the shell. The first `interceptAll` parameter, `mi`, will be the object that describes method `m` of object $\mathcal{Q}$ — there will always exist one.

Inside `interceptAll`, method `m` of $\mathcal{Q}$ can be called as

```
mi.invoke(vetArg)
```

Method `invoke` will call the method described by `mi` using parameters taken from `vetArg`.

`interceptAll` allows shells to have the full functionality of metaobjects although with some overhead. See [4] for more details.

12.8 Dynamic Extension

Methods of a single object can be replaced by methods of a shell. Methods of a class, which includes methods of all objects of the class, can be replaced by methods of a dynamic extension class. The syntax for dynamic extensions is identical to the syntax for shell classes. In fact, a dynamic extension class can be used as shell class and vice-versa. This means dynamic extension classes:

- can inherit from other extension classes;
- method `interceptAll` also works with extension classes;
- cannot have associate class objects. If there is a dynamic extension class `Border`, there cannot be a class object `Border`.

Figure 12.6 shows the dynamic extension `Border` that can be attached to class `Window` or its subtypes. The programmer should specify at compile time the classes to which she wants to attach dynamic extensions. These classes compose the "allowed set" for a given extension class. Different from [4], it is not necessary to declare class `Window` using "`reflective(extension)`".

An abstract class may belong to the "allowed set" of an extension class. However, the extension class should not define any method `m` such that `m` of the abstract class is declared as abstract. By the semantic rules of Green, an abstract method will never be called. Then the extension method that replaces `m` will not be called either. This is not an error although the compiler should issue a warning.

To attach extension `Border` to class `Window` one should write

```
Meta.attachExtension(Window, Border)
```

`Meta` is a module and this compile-time message send will be replaced by a message send to be executed at run time. If there is not sufficient memory an exception `OutOfMemoryException` will be thrown. If `Window` is not in the allowed set of `Border` there will be a compile error.

If `Border` declares instance variables, these should be attached to all `Window` objects. This is made on demand: when an object is going to use its `Border` instance variables, the run-time system makes a test to discover if memory for these variables have been allocated. If not, it allocates them. A hash table with pairs (`obj`, `shellObj`) is used to associate `Window` objects with theirs `Border` instance variables. To object `obj` has been conceptually added the variables of object `shellObj`. Object `shellObj` has only the instance variables declared in `Border`. Using the object address as key, one can retrieve the object instance variables.

```
class Window
  proc init(...)
    begin
    ...
    end

  public:
    proc draw()
      ..
    ..
end

  // same syntax as shell classes

shell class Border(Window)
  public:
    proc draw()
      begin
      self.drawBorder();
      super.draw();
      end
  private:
    proc drawBorder()
      ...
end
```

Figura 12.6: Dynamic extension `Border` to subtype-`Window` classes

The test to discover if memory was allocated for the `Border` instance variables is added at the beginning of each `Border` method that accesses instance variables.

Suppose shell class `Border` has a parameterless constructor. When an extension `Border` is attached to class `Window`, the constructor should be called on each `Window` object to possibly initiate `Border` instance variables of each object. But that would be very expensive because all objects of `Window` should be found. An alternative solution would be to call the constructor the first time a `Window` object receives a message after the attachement of `Border` to `Window`. The implementation for this would be complex and would have a large memory overhead. A method that calls the constructor should be created for each `Window` method.

A third solution would be to call the constructor only when the extension instance variables are allocated. Then the constructor could initiate these variables. This is our choice. In the beginning of each method that accesses extension instance variables there is code to allocate memory for the instance variables (if this has not happened yet) and to call the extension constructor.

Note the semantics of extension constructors is different from the semantics of normal constructors. An extension constructor is only called when the extension instance variables are created, which may occur at any time during the object lifetime. Because we do not know when the extension constructor will be called, it should be only used to initiate the extension instance variables.

An extension is removed from a class by the command

```
Meta.removeExtension(Window)
```

which will be replaced by a message send that may throw exception `NoExtensionException`. This will occur if no extension is attached to class `Window`. More than one extension may be attached to a class. They are attached and removed in a stack fashion.

An example of use of dynamic extensions is given next.

```
var w1, w2 : Window;

w1 = Window.new(...);
w2 = Window.new(...);
  // draw windows without borders
w1.draw();
w2.draw();

Meta.attachExtension( Window, Border );

  // draw windows with borders
w1.draw();
w2.draw();


var catch : CatchMetaException;

catch#init();
try(catch)
  Meta.removeExtension(Window);
    // draw windows without borders
  w1.draw();
  w2.draw();
end
```

All message sends to all objects of a class can be intercepted by declaring a method `interceptAll` in an extension class that is then attached to the class. This mechanism is used with dynamic extensions as it is with shells. The semantics is the same.

Capítulo 13

Standard Class Objects

13.1 Introduction

A Green program does input, output, and controls itself through objects `In`, `Out`, `OutError`, `Memory`, `Runtime`, `Screen`, and `Storage`. A brief description of each object follows.

- `In` is used for input from the standard input.
- `Out` is used for output to the standard output.
- `OutError` is used for output to the standard error output.
- `Storage` abstracts the file system and the Internet, although methods for working with the Internet have not been defined yet.
- `Memory` has methods that control memory use of the whole program. One can turn on/off the garbage collector, ask the number of bytes of free memory available, and so on.
- `Runtime` abstracts (part of) the program run-time system. This object has a method for finishing the program and a method that returns an object describing the method call stack at run time.
- `Screen` is used for graphical handling and user interaction through devices like keyboards and mice. The specification of `Screen` is troublesome and beyond the capacity of the author of this report. Then hopefully `Screen` will be defined by someone else.

Objects `Memory` and `Runtime` interact only with the program. Objects `In`, `Out`, `OutError`, `Storage`, and `Screen` make the linking between the program and the Operating System. This linking is so important that it deserves to be better studied, which is made next.

Green assumes its programs will run in a fictitious Object-Oriented Operating System called Rainbow. Although Rainbow does not exist, the features described below may be simulated by

- a Green interpreter if the programs are translated into instructions of some virtual machine;
- a run-time system manager. That is, a Green program would be compiled not to a ready-to-run executable program but to a format that needs to be executed by other program, the run-time system manager:

```
C:\>manager aProgram parameters
```

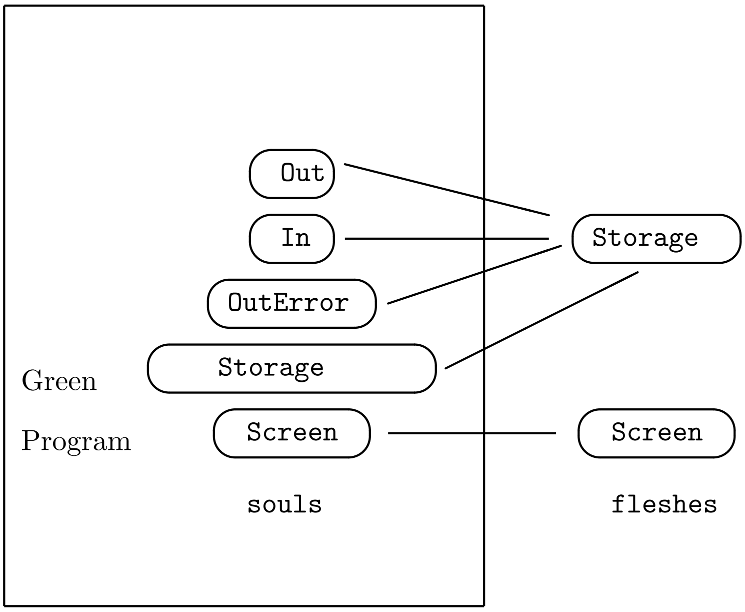

Figura 13.1: Souls and fleshes in a Green Program

So `aProgram` would be controlled[1] by `manager` as it is by the Operating System. The symbols "`C:\>`" are the DOS prompt.

13.2 Rainbow

Rainbow considers every executable program in disk or some other storage device as a *class*. The loading and execution of a disk program is carried out through the following steps:

- the class is loaded to the main memory and one object of it is created. This object will represent an executing process;
- Rainbow creates objects `Storage` and `Screen` that match the objects with the same names of the Green program. These Rainbow objects are called *flesh objects*. The corresponding objects of the Green program are called *souls* — see Figure 13.1. Each *soul* has a reference to the *flesh* object with the same name. In addition to that, soul objects `In`, `Out`, and `OutError` have references to flesh object `Storage`. After all, input and output to standard devices are only file manipulation. Each method of a soul object just forwards the message to the corresponding flesh object; that is, to the flesh object the soul object references.
- each executable program in disk, which is just a class, may have a set of associate shells to the program itself and to each of the flesh objects `Storage` and `Screen`. These shells are created and attached to the flesh objects and possible to the program itself;
- message `run` is sent to the object representing the program passing as parameter the program parameters.

[1]To a certain extent, of course, since `aProgram` could bypass the run-time manager.

This model gives us freedom to modify a program by associating shells to it using Rainbow. Some interesting things can be done:

- one can easily attach shells connecting the output of a program (flesh object `Out`) to the input (flesh object `In`) of another, thus implementing a Unix pipe;
- a shell may be attached to flesh object `Screen` such that all graphical output (windows, buttons, menus, etc) is done not only to the screen associate to the program but also to the screen of another computer in the same network;
- a Rainbow shell may be attached to flesh object `Storage` to prevent an untrusted program from writing to or deleting disk files. This shell would play a rôle similar to the Java Virtual Machine when executing applet code.

The possibilities of program control are endless.

A future feature of Green would allow any class object of the program to be associated with a flesh object. Then the program behavior could be easily modified by putting shells in its flesh objects. Of course, the program can attach shells to its own soul objects (`In`, `Out`, ...) thus modifying the program behavior in the same way as attaching shells to flesh objects. Note a program has no permission to

access flesh objects (besides sending messages to them) and Rainbow cannot access soul objects.

Objects `Memory` and `Runtime` may have flesh objects in the future. Flesh objects `Memory` and `Runtime` would have a subset of the methods of the corresponding soul objects. If flesh `Memory` had all methods of soul `Memory`, Rainbow could, for example, turn off the garbage collection, which does not make sense. But Rainbow could furnish more memory for the program through

flesh object `Memory` or inspect the program run-time stack through flesh object `Runtime`.

13.3 The Class Objects

All objects described in this chapter are detailed below. Note:

- flesh objects have the same interface as the soul objects with the same name;
- object `Screen` is not described — its definition demands knowledge about low level aspects of screen/mouse/keyboard interactions that the author of this languages does not have;
- methods will be added to most objects described next. Their interfaces nowadays is minimal and need improvements. For example, object `Storage` has few methods for file handling and no for Internet.

Here are the objects. Only the method interfaces are shown.

```
object In
  private:
      // reads from the standard input
    proc readCh()      : char
    proc readByte()    : byte
    proc readInteger() : integer
    proc readLong()    : long
    proc readReal()    : real
    proc readDouble()  : double
```

```
    proc readString()  : String
    proc readLine()    : String
end


object Out
  public:
      // print an array of objects. Any type is allowed
    proc write( v : ... array(Any)[] )
      // print a newline after each object
    proc writeln( v : ... array(Any)[] )
end


object OutError
  public:
      // print an array of objects. Any type is allowed
    proc write( v : ... array(Any)[] )
      // print a newline after each object
    proc writeln( v : ... array(Any)[] )
end



object Memory
  public:
      // returns the size in bytes of the largest memory block
    proc sizeLargestBlock() : long
      // returns the size in bytes of the free memory
    proc sizeFreeMemory() : long
      // calls the garbage collector
    proc doGarbageCollection()
      // turns on and off the garbage collection
    proc collectionOn()
    proc collectionOff()
end



object Storage
  public:
    proc removeFile( fileName : String ) : boolean
    proc renameFile( oldName, newName : String ) : boolean
      // opens a file and returns a file descriptor
    proc openFile( fileName : String ) : integer
      // closes a file with file descriptor fd
    proc closeFile( fd : integer )
      // reads n bytes from file fd to array in
```

```
    proc read( fd, n : integer; in : array(byte)[] )
      // write n bytes from out to file fd
    proc write( fd, n : integer; out : array(byte)[] )
      /* returns an error code. This method should be called after file
         or Internet operations. The values returned  are given in the
         enumerate declaration that follows. */
    proc getError() : integer

    enum( fileDoNotExist_e, cannotClose_e, smallArraySize_e,
          unknownFileDescriptor_e, readOnlyFile_e, internalError_e );

end


/* The methods of Storage are used to implement class BasicStream and
   its subclasses used for file manipulation. These classes are described
   elsewhere. */


object Runtime
  // this object represents the run-time system

  public:
    proc exit( errorCode : integer )
      /* finishes the program and returns errorCode. In an object-oriented
         operating system an object would be returned */
    proc putAtEndList( f : Function )
      /* puts f in a list of Function objects. At the end of the
         program, message exec will be sent to all objects of this
         list. Class Function belong to the Green Standard Library. */
end
```

Appendix B describes some methods added to `Runtime` that allows one to inspect the method call stack and the stack of exception catch objects. See Chapter 15 for the definition of class `Function`.

Capítulo 14

Parameterized Classes

Parameterized classes are implemented using the compile-time metaobject protocol (MOP), topic not presented in this report. However, their use will follow nearly the syntax described in the following paragraphs, which will be changed as soon as the MOP is defined.

The language supports a simple form of parameterized classes whose syntax is shown in Figure 14.1. Parameters are specified in a semicolon separated list delimited by round brackets, following the class name in the header. Each parameter can be in one of following forms.

- `T : A`

  where `T` is the parameter name. Real parameters must be ext-subtypes of `A`. A class `S` is an ext-subtype of class `T` if `S` is subtype of `T` and `S` defines all `init` methods `T` defines. That is, `init` methods belong to a class type in the definition of ext-subtype. Class `A` must be a class. That is, `A` cannot be an expression "`type(X)`" in which `X` is a class object.

- `T`

  where `T` is the parameter name. The corresponding real parameter may belong to any type, including basic classes.

The round brackets with the parameters should be put in the class *or* in the class object, whichever comes first. So, we can declare

```
object List( T : B )
  ...
end

class List
  ...
end
```

```
class A( parameter-list )
    // constructors
  public:
    // methods
  private:
    // methods and instance variables
end
```

Figura 14.1: Syntax for parameterized classes and use `List` as in

```
var list : DS.List(Person);
```

The compiler will check if `Person` is ext-subtype of `B`. If not, an error is issued. Inside class `list`, a `T` object may be allocated. The parameters to `new` follow the rules for class `B`. That is, if `B` defines a method

```
init( x : integer; ch : char )
```

then inside `List` an expression

```
T.new(3, '#A')
```

is legal.

`DS` is the module in which `List` is. To furnish the real parameters to a parameterized class, as in "`List(Person)`", is called "the instantiation of class `List`". When the compiler finds this instantiation, it replaces the formal by the real parameters in the body of class `List`, creating a new class "`List$p$1$Person`". Then the compiler compiles this class, which may have errors.

It is pretty common to design a parameterized class in which the type parameter may be a basic class or a subclass of `Any`. That is, the type parameter may be a class with value or reference semantics. This creates a problem because the operators (`< + *` ...) of basic classes cannot be declared in normal classes therefore dividing the world of types in two: those who have operators and those who do not. To see why this is important, we will study the following parameterized class.

```
class List(T)
  public:
    proc getGreatest() : T
      ...
    proc add( x : T )
    ...
end
```

Method `getGreatest` returns the greatest element in the list. If `T` is a basic class, this method should use operators like `<` or `>=` to do its job. If `T` is a subclass of `Any` (reference class), `getGreatest` should use a method like `lessThan` or `greaterEqual`. Then the `List` class could be used either with basic classes or reference classes.

To prevent such problem, the compiler[1] will replace the comparison operator by message sends whenever the real parameter `T` to the parameterized class is a subclass of `Any` (`T` is a normal class). Then, if `x` and `y` belong to type `T`,

```
a > b
```

will be replaced by

```
a.greaterThan(b)
```

Methods `equals`, `notEqual`, `lessThan`, `greaterThan`, `lessEqual`, and `greaterEqual` will replace `==`, `<>`, `<`, `>` `<=`, and `>=`, respectively. The arithmetic operators will not be replaced. Their semantics are more complex than that of comparison operators.

Parameterized classes may be implemented either with code sharing or code duplication. In the first case there will be a single code for a parameterized class `List` regardless of the number of instantiations of it. In the second case, the compiler will create a different class each time `List` is instantiated with a new class: `List(integer)`, `List(Person)`, `List(Figure)`.

[1]Assume that. In fact, parameterized classes are not supported by Green. They are implemented using the MOP that still needs to be defined.

The official semantics of Green parameterized classes requires code duplication. If code sharing is used, the compiler creator should take care of the relationship between dynamic extensions and parameterized classes. We may describe this caveat using an example.

Class `List` has method `print` to print the list. Class `Slist` inherits from `List` and redefines method `print`, which calls `List::print` through `super`:

```
class SList(T) subclassOf List(T)
  public:
    proc print()
      begin
      ...
      super.print();
      end
    ...
end
```

Both classes have a parameter `T`, the list element type. Suppose now an extension is attached to `List(Person)` changing its `print` method. How would `SList(Figure)::print` behave since this method has a call "`super.print()`", which should call `List(Figure)::print` and `print` was replaced in `List(Person)` ?

`SList(Figure)` inherits from `List(Figure)` and should not be affected by an extension on `List(Person)`. But if parameterized classes are implemented using code sharing, `List(Person)::print` and `SList(Figure)::print` are the same method. And a change in one should reflect in the other.

A solution to this problem would be to duplicate the code whenever there is a call to super and the superclass is parameterized.

Capítulo 15

The Standard Library

Green has a standard library with data structure and file handling classes. Of course, the following classes will be improved and a lot of new classes will be added to the standard library.

File Handling

```
object BasicStream
  public:
    enum (read_s, write_s, readwrite_s, append_s);
end


abstract class BasicStream
  public:

    abstract proc open( name : String; mode : integer )
                   ( exception : CatchFileException )
      /* If the name is not valid, calls
              raise(InvalidNameException) : String
         of variable exception.
         The return value should be a new file name with which open
         tries to open again. If the operation fails, raise is again called
         and the method tries to open the file. This repeats till a
         valid name is given or raise throws an exception.
             An existing file is open only for reading is mode read_s
         is used. If the file does not exist, method
                   raise(NonExistingFileException) : String
         of variable exception is called till a valid name is given
         or raise throws an exception
         In any other error, method throw(OpenFileException) is called.
      */

    abstract proc close()
                    ( exception : CatchFileException )
      // In error, calls method raise(CloseFileException) or
```

```
      // raise(FileIsClosedException)

    abstract proc getSize() : integer
                         ( exception : CatchFileException )
    abstract proc getName() : String
end


class InputStream subclassOf BasicStream
  proc init()

  public:

    proc read( v : array(char)[]; n : long )
             ( exception : CatchFileException )
      // In error, calls raise(ReadFileException)

    proc read( v : array(byte)[]; n : long )
             ( exception : CatchFileException )

    proc read( s : DynString )
             ( exception : CatchFileException )

    proc readln( s : DynString )
               ( exception : CatchFileException )
end


class OutputStream subclassOf BasicStream
  proc init()

  public:

    proc write( v : array(char)[] )
              ( exception : CatchFileException )
      // In error, calls raise(WriteFileException) or
      // raise(WriteFileInterException)

    proc write( v : array(char)[]; n : long )
              ( exception : CatchFileException )

    proc write( v : array(byte)[] )
              ( exception : CatchFileException )

    proc write( v : array(byte)[]; n : long )
              ( exception : CatchFileException )

    proc write( s : DynString )
```

```
              ( exception : CatchFileException )

    proc writeln( s : DynString )
                ( exception : CatchFileException )
end



class Stream subclassOf BasicStream
  proc init()

  public:
    // all methods of InputStream and OutputStream
    ...
end


class OpenFileException
  proc init( f : File )
  public:
    proc getFile() : File
end

// there is a class equal to the above for every every file exception.
// There is a throw method in class CatchFileException for each file
// exception

object CatchFileException
  proc init()
    begin
    singleInstance = CatchFileException.new(' ');
    end

  public:
    proc new() : CatchFileException
      begin
      return singleInstance;
      end
end


class CatchFileException
  proc init( x : char )
    begin
      // fake. Just to cause the creation of a new(char) method in
      // the class object
    end
```

```
  public:
    proc throw( exc : InvalidNameException )
    proc throw( exc : NonExistingFileException )
      // file does not exist
    proc throw( exc : OpenFileException )
      // file cannot be openend
    proc throw( exc : OpenReadOnlyFileException )
      // attempt to open for writing a read-only file
    proc throw( exc : ReadFileException )
      // attempt to read more bytes than the available or another
      // read error
    proc throw( exc : WriteSharedFileException )
      // two processes are trying to write to the same file
    proc throw( exc : WriteFileException )
      // no space for writing or other write error
    proc throw( exc : CloseFileException )
      // cannot close file
    proc throw( exc : FileIsClosedException )
      // attempt to use a non-open file

    proc raise( exc : InvalidNameException )
              ( exception : CatchFileException ) : String
    proc raise( exc : NonExistingFileException )
              ( exception : CatchFileException ) : String
    proc raise( exc : OpenFileException )
              ( exception : CatchFileException ) : String
    proc raise( exc : OpenReadOnlyFileException )
              ( exception : CatchFileException ) : String
    proc raise( exc : ReadFileException )
              ( exception : CatchFileException )
    proc raise( exc : WriteSharedFileException )
              ( exception : CatchFileException )
    proc raise( exc : WriteFileException )
              ( exception : CatchFileException )
    proc raise( exc : CloseFileException )
              ( exception : CatchFileException )
    proc raise( exc : FileIsClosedException )
              ( exception : CatchFileException )
end
```

The methods of all classes for file manipulation uses methods `raise` for throwing exceptions. Methods `raise` of `CatchFileException` will just throw the exception they receive as parameters. Subclasses of `CatchFileException` can take a different approach. For example, subclass `CatchFileExceptionUser` tries to get help from the user when methods

raise(InvalidNameException)@

and

raise(NonExistingFileException)@

are called. The user can then suggest other file name and retry the operation. Subclass `CatchFileExceptionEnd` prints an error message and terminates the program — a radical approach.

Data Structure Classes

The following classes are parameterized classes and described in Chapter 14. They should be used with the prefix "`DS`" as in

```
var list  : DS.List(Person)
var queue : DS.Queue(double);
```

Class `Iter` is an iterator class. All data structure classes have a method that return an iterator object which belong to a subclass of `Iter`. All classes inherit from `Container`. Contrary to other object-oriented libraries, we did not do a deep inheritance hierarchy for data structure classes. Not only this is unnecessary but also dangerous: a class may be declared superclass of another without being semantically subclass of it. This seems to be pretty common in hierarchies of data structure classes.

```
abstract class Container( T : Any )
    proc init()
      begin
      index = -2;
      end
  public:
    abstract proc add( elem : T )
                     ( exception : CatchOutOfMemoryException )
      // inserts elem in the container

    abstract proc get() : T
                     ( exception : CatchNotFoundException )
      /* gets one element from the container which is not removed
        from it. */

    abstract proc remove() : T
                     ( exception : CatchNotFoundException )
      // removes one element from the container and returns it.

    abstract proc removeAll()
      // empties the container

    abstract proc getSize() : integer
      // returns the number of elements in the container

    abstract proc empty() : boolean
      // returns true if there is no element in the container

    abstract proc full() : boolean
      /* returns true if there is no more space in the container. This
         method only returns false in static containers whose number of
         elements is defined at the creation of the container */
```

```
abstract proc getIter() : DS.Iter(T)
  // returns an iterator for the container

abstract proc forEach( f : Function(T) )
  // calls "f.exec(x)" for each element x of the container

abstract proc replaceBy( cmd : Command(T) )
  // replaces each element x in the container by "cmd.doIt(x)"

abstract proc collect( f : Filter(T) ) : Container(T)
  /* collects all container elements x such that "f.test(x)"
     evaluates to true. These elements are inserted in a new
     container that is returned by this method. */

abstract proc  remove( f : Filter(T) )
  /* removes all array elements x from the container whenever
     "f.test(x)" evaluates to true. */

proc reset()
  /* reset, next, and endIter compose a lightWeight iterator. It
     should be used as in the example:
        list.reset();
        while (elem = list.next()) <> nil do
          Out.writeln(elem);
        list.endIter();
     This iterator can only be used when T is a reference class
     (not a basic class like char, byte, etc) and nil cannot be
     inserted in the container. Method next returns nil when there is no
     more elements.
        Method endIter should be called after the use of the
     iterator. If it is not called, the next call to reset will cause a
     run-time error:
         list.reset();
         x = list.next();
           // run-time error in the following line
         list.reset();
     This prevents one of calling reset in the middle of an use of
     the iterator:
         list.reset();
         while (x = list.next()) <> nil do
           begin
           select( list, x );
           ...
           end
     If select calls ``reset'' on list, there will be an
     error. Only one scanning of the container can be made at a time with
     this kind of iterator.
```

```
          This type of iterator does not need a separate object of
         type DS.Iter(T) thus saving memory and time of the garbage
         collector.

         This method should be implemented as
          begin
          if index <> -2
          then
             Out.writeln("Iterator error");
             Runtime.exit(1);
          else
             index = -1;
          endif
          end
      */


    abstract proc next() : T

    abstract proc endIter()
      begin
      index = -2;
      end

    proc toArray() : array(T)[]


end


abstract class Iter( T : Any )
  public:
    abstract proc more() : boolean;
    abstract proc next() : T
      assertion
        before more();
      end
    abstract proc reset()
      // begin everything again
    abstract proc toArray() : array(T)[]
      // returns all elements in an array
end


class IterFilter( T : Any )
    proc init( piter : Iter(T); pf : Filter(T) )
      begin
      iter = piter;
```

```
      end
  public:
    proc more() : boolean
      begin
      return iter.more();
      end
    proc next() : T
        assertion
          before more();
        end
      begin
      var item : T;
      loop
        if not self.more()
        then
          return nil;
        else
          item = iter.next();
          if f.test(item)
          then
            return item;
          endif
        endif
      end
      end

    proc toArray() : array(T)[]
      begin
      return iter.toArray();
      end

    proc reset()
      begin
      iter.reset();
      end

  private:
    var iter : Iter(T);
        f    : Filter(T);
end


abstract class Filter( T : Any )
  public:
    abstract proc test( x : T ) : boolean
end
```

```
abstract class Command(T)
  public:
    abstract proc doIt( x : T ) : T
end


abstract class Function(T)
  public:
    abstract proc exec( x : T )
end


/* In the classes below, a D before the class name means the class is
   a Dynamic container. That is, a class DList is better than a class
   List when the number of list elements varies greatly during the list
   lifetime. Possibly DList is implemented using a linked list and List
   using an array. So DList and the like will produce a lot of objects
   causing memory and speed overhead (because of the garbage
   collector).
*/


class List( T : Any ) subclassOf Container(T)

  /* A list of elements. There is no order to insert or remove
     elements.
     This class is better when the number of list elements does not
     vary greatly during the list lifetime. */

  public:
    // all Container(T) methods are defined but no new one is added
    ...
  private:
    ...
end


class DList( T : Any ) subclassOf Container(T)
  /* A list of elements. There is no order to insert or remove
     elements.
     This class is better when the number of elements
     varies greatly during the container lifetime. */

  public:
    // all Container(T) methods are defined but no new one is added
    ...
  private:
    ...
```

```
end



class Stack( T : Any ) subclassOf Container(T)
  /* A stack of elements.
     This class is better when the number of elements does not
     vary greatly during the container lifetime. */

  public:
    // all Container(T) methods are defined but no new one is added
    ...
  private:
    ...
end


class DStack( T : Any ) subclassOf Container(T)
  /* A stack of elements.
     This class is better when the number of elements
     varies greatly during the container lifetime. */

  public:
    // all Container(T) methods are defined but no new one is added
    ...
  private:
    ...
end



class Queue( T : Any ) subclassOf Container(T)
  /* A queue of elements.
     This class is better when the number of elements does not
     vary greatly during the container lifetime. */

  public:
    // all Container(T) methods are defined but no new one is added
    ...
  private:
    ...
end


class DQueue( T : Any ) subclassOf Container(T)
  /* A queue of elements.
     This class is better when the number of list elements
     varies greatly during the container lifetime. */
```

```
  public:
    // all Container(T) methods are defined but no new one is added
    ...
  private:
    ...
end



class DoubleQueue( T : Any ) subclassOf Container(T)
  /* A double queue of elements. Insertion and removal of elements can
     occur in both sides of the queue.
     This class is better when the number of list elements does not
     vary greatly during the container lifetime. */

  public:
    /* all Container(T) methods are defined and some ones are added.
       Method put, get, and remove are equivalente to putBack,
       getFront, and removeFront. */
    ...
    abstract proc addFront( elem : T )
                          ( exception : CatchOutOfMemoryException )
      // inserts elem in the front of the queue

    abstract proc addBack( elem : T )
                         ( exception : CatchOutOfMemoryException )
      // inserts elem in the back of the queue

    abstract proc getFront() : T
                          ( exception : CatchNotFoundException )
      /* gets one element from the front of the queue which is not removed
         from it. */

    abstract proc getBack() : T
                         ( exception : CatchNotFoundException )
      /* gets one element from the back of the queue which is not removed
         from it. */

    abstract proc removeFront() : T
                             ( exception : CatchNotFoundException )
      // removes one element from the front of the queue and returns it.

    abstract proc removeBack() : T
                            ( exception : CatchNotFoundException )
      // removes one element from the back of the queue and returns it.
```

```
  private:
    ...
end


class DDoubleQueue( T : Any ) subclassOf Container(T)
  /* A double queue of elements. Insertion and removal of elements can
     occur in both sides of the queue.
     This class is better when the number of elements
     varies greatly during the container lifetime. */

  public:
    // the same methods as DoubleQueue

    ...
  private:
    ...
end


abstract class HashFunction( T : Any )
  public:
    proc setSize( psize : integer )
      begin
      size = psize;
      end
    abstract proc hash( elem : T ) : integer
  private:
    var int size;
end


class Compose( T, U : Any )
    proc init( pt : T; pu : U )
      begin
      t = pt;
      u = pu;
      end

  public:
    proc getT() : T
      begin
      return t;
      end

    proc getU() : U
      begin
      return u;
```

```
      end
  private:
    var t : T;
        u : U;
end


class Dict( KeyType, ValueType : Any )
    proc init()
    proc init( pSize : integer )
      // pSize is the maximum estimated number of elements in the table
    proc init( pSize : integer; phashFunction : HashFunction(KeyType) )
      // phashFunction is the hash function for type KeyType

  public:
    proc add( key : KeyType; value : ValueType )
            ( exception : CatchOutOfMemoryException )
    proc get( key : KeyType ) : ValueType
            ( exception : CatchNotFoundException )
    proc remove( key : KeyType ) : ValueType
               ( exception : CatchNotFoundException )
    proc getSize() : integer
      // number of elements in the dictionary
    proc removeAll()
    proc getIter() : DS.DictIter(KeyType, ValueType)
  private:
    ...
end


class DictIter( KeyType, ValueType : Any )
    proc init( pdict : Dict(KeyType, ValueType) )

  public:
    proc more() : boolean
    proc next() : Container(KeyType, ValueType)
  private:
    ...
end


class IntegerSet
    proc init()

  public:
    proc add( n : integer )
            ( exception : CatchOutOfMemoryException )
    proc remove( n : integer )
```

```
              ( exception : CatchNotFoundException )
    proc inSet( n : integer ) : boolean
    proc empty() : boolean
    proc getIter() : DS.Iter(integer)
    proc removeAll()
  private:
    ...
end


object Vector( T : Any )
  public:
    const defaultSizeVector = 20;
end


  // since object Vector has a parameter, class Vector must have too.

class Vector( T : Any )  subclassOf  Container(T)
    proc init( p_size : integer )
      /* size of the vector. Initially the number of elements is
         zero. That is,  getSize() returns 0. */

    proc init()
      // uses defaultSizeVector to initiate the object
      begin
      init(defaultSizeVector);
      end


  /* an high-level array. Method
        get() : T
     get the last array element and
        add( elem : T )
     inserts elem after the last array element. So Vector works
     much like a stack. The maximum number of elements is given by
     getMaxSize(). However, the number of elements that have effectively
     been put in the vector is given by getSize(). If one tries to put
     an element in a position after the end of the vector (after position
     getMaxSize()), all positions between the last and this new end is
     fulfilled with nil. For example, in

         var v : Vector(Integer);
         v#init();
         v.add( Integer.new(5) );
         v.add( Integer.new(7), 2 );
         Out.writeln( v.get(1) );
```

```
    the position 1 of the vector v is initiated to nil since the
    statement
         v.add( Integer.new(7), 2 )
    bypass position 1.


       Vector is a dynamic vector. It grows and shrinks as needed.

  */
  public:


    proc add( elem : T )
            ( exception : CatchOutOfMemoryException )
      // inserts elem after the last array element

    proc add( elem : T; i : integer )
            ( exception : CatchOutOfMemoryException )
      // inserts elem in the position i. See observations above.


    proc get() : T
            ( exception : CatchNotFoundException )
      /*  gets one element from the container which is not removed
         from it. */

    proc get( i : integer ) : T
            ( exception : CatchNotFoundException )
      /* gets the i-th element from the vector which is not removed
         from it. */

    proc remove() : T
               ( exception : CatchNotFoundException )
      // removes one element from the container and returns it.

    proc remove( i : integer ) : T
               ( exception : CatchNotFoundException )
      /* removes the i-th element from the vector and returns it.  All
         other elements are left-shifted. Exception NotFoundException is
         thrown if i < 0 or i >= getNum() */

    proc getSize() : integer
      // returns the real number of vector elements
    proc getMaxSize() : integer
      // returns the maximum number of vector elements
    proc toArray() : array(T)[]
end
```

Capítulo 16

Introspective Reflection

The introspective reflection library is a set of objects and classes that allow a program to access information about itself at compile or run time. For example, the program can know the class of an object, the methods and instance variables of that class, the parameter names and types of each method, and so on.

There is a build-in object called `GreenCompiler` whose methods are executed at compile time. This object has the following methods:

- `getType( id : Symbol ) : TypeNo`
  which returns the type the identifier `id` was declared with if `id` is a variable of returns the set of method signatures of `id` if it is a class object. `TypeNo` is the class of the Abstract Syntax Tree used to represent type. this method is identical to function `type` defined in Section 10.1;

- `getCompilerName() : LiteralString`
  which returns the compiler name;

- `getCompilerVersion() : LiteralString`
  which returns the compiler version;

- `getStrDate() : LiteralString`
  which returns a string with the date of compilation in the format "mm/dd/yyyy".

- `getStrTime() : LiteralString`
  which returns a string with the hour/minute/second of compilation in the format "hh:mm:ss";

- `getFileName() : LiteralString`
  which returns a string with the name of the file being compiled, if there is one being compiled the moment this method is executed. The compiler may create Green code that is not associate to any particular file. In this case `getFileName` returns an empty string;

- `getStrCurrentClass() : LiteralString`
  returns a string with the name of the class being compiled. Inside either class `Person` or class object `Person`, this method would return "`Person`". Returns an empty string if no class is being currently compiled;

- `getStrCurrentMethod() : LiteralString`
  returns a string with the name of the method being compiled, if there is one. Returns an empty string if no method is being currently compiled;

- `getCurrentClass() : TypeNo`
  returns a symbol that represents the current class. Then, inside a method of a class-`Person` (or class object `Person`), the declaration
  ```
  var p : GreenCompiler.getCurrentClass();
  ```
  is equivalent to
  ```
  var p : Person;
  ```
  If there is no current class or class object, a compile error is issued;

- `getCurrentMethodInfo() : MethodInfoNo`
  returns an object with information about the current method being compiled. That is, the command
  ```
  mi = GreenCompiler.getCurrentMethodInfo();
  ```
  of a parameterless method `m` may be replaced by
  ```
  mi = GreenCompiler.getCurrentClass().getAssociateClassInfo().getMethod("m", nil);
  ```
  There is an error if there is no current method;

- `getEofOfList( aClass : ClassNo ) : LiteralNo`

  returns a literal value that is generally used as end-of-list[1] (EOL) mark for list of objects of class `aClass`. `ClassNo` is a class of the Abstract Syntax Tree of the compiler that represents a class.

  All subclasses of `Any` use `nil` as EOL. Classes `integer`, `long`, `real`, and `double` use `-1`. `byte` uses `255`. Class `char` uses `'\0'`. Class `boolean` cannot be parameter to this method — that would not make sense.

  An example of use of this method is shown below in which all array elements are valid till `-1`. Of course, one could not have put `-1` as a valid array number.

  ```
  var v : array(integer)[];
  var i : integer;
  ... // initialize v
  i = 0;
  while v[i] <> -1 do
    begin
    Out.writeln( v[i] );

    ++i;
    end
  ```

  The `getEofOfList` method was devised to be used in parameterized classes in which this code would be generalized to

  ```
  class List(T)
    public:
      ...
      proc write()
        begin
        var i : integer;
  ```

[1]The method pretends to return an end-of-file mark.

```
ClassInfo
    ValueClassInfo
    RefClassInfo
        AbstractClassInfo
        ConcreteClassInfo
            NormalClassInfo
            ArrayClassInfo
```

Figura 16.1: Hierarchy of `ClassInfo`

```
          i = 0;
          while v[i] <> GreenCompiler.getEofOfList(T) do
            begin
            Out.writeln( v[i] );
            ++i;
            end
          end
      ...
      private:
          var v : array(T)[];
    end  // List(T)
```

Class `LiteralString` belongs to the Abstract Syntax Tree of the compiler and represents values of class `String`.

The hierarchy of classes that describe the program classes at run time is shown in Figure 16.1. Each class in the system is described by an object of `ClassInfo` or one of its subclasses. To get the object of `ClassInfo` that describes a class `Person` we do

```
var ci : ClassInfo;

ci = Person.getAssociateClassInfo();
  // print "Person"
Out.write( ci.getName() );
```

The class of an object can be got by calling method `getClassObject`:

```
var p : Person;
...
var any : AnyClassObject;
any = p.getClassObject();
Out.writeln( any.getAssociateClassInfo().getName() );
```

The methods of a class are got in the following way:

```
var ci : ClassInfo;
var p : Person;
...
ci = p.getClassInfo();
```

```
MethodInfo
    ClassMethodInfo
    ObjectMethodInfo


InstanceVariableInfo
    ClassInstanceVariableInfo
    ObjectInstanceVariableInfo


AnyObjectInfo
    ObjectInfo
    ClassObjectInfo
```

Figura 16.2: Hierarchy of the introspective reflection library

```
var v : array(ClassMethodInfo)[];
v = ci.getMethods();
for i = 1 to v.getSize() - 1 do
  Out.writeln( v[i].getName() );
```

Instead of using the `ClassInfo` hierarchy to get information about classes, one can access information about individual objects, which includes class objects:

```
var ac : Account;
...
var objInfo : AnyObjectInfo;
objInfo = ac.getInfo();

  // list the names of all instance variables of the object
var v : array(ObjectInstanceVariableInfo)[];
v = objInfo.getInstanceVariables();
for i = 0 to v.getSize() - 1 do
  Out.writeln( v[i].getName() );

var objMethodInfo : ObjectMethodInfo;
  // get info about method "getBalance" of object ac
objMethodInfo = objInfo.getMethod("getBalance");
  // calls method "getBalance" of object ac. The balance returned is printed
Out.writeln( objMethodInfo.invoke(nil) );
```

Note the exceptions this code can generate are not considered.

The hierarchies of `MethodInfo`, `InstanceVariableInfo`, and `AnyObjectInfo` are shown in Figure 16.2.

One can create and use an array whose element type will only be known at run time without using the reflection library. This is possible because

all arrays are subclasses of class `AnyArray`. See the following code.

```
var any : AnyClassObject = integer;
```

```
var anArray : AnyArray;
  // creates a 10-element integer array
anArray = array(any)[].new(10);
anArray.set(5, 0); // anArray[0] = 5;
anArray.set(3, 1); // anArray[1] = 3;
Out.writeln( anArray.get(0) ); // print 5
```

If `any` does not refer to a class object at run time, an exception will be thrown. All the classes of the Introspective Reflection Library are described in Appendix B. Note a normal programmer cannot create objects of these classes.

One should take care in using the classes of the Introspective Reflection Library because there are objects describing these same classes. It is easy to take information on a class `Person` for information on the class of the object that describes class `Person`. As an example, the following code will write "`ClassInfo`", which is the name of the class of `ci` at run time.

```
var ci : ClassInfo;

ci = Person.getAssociateClassInfo();
Out.writeln( ci.getClassInfo().getName() );
```

The type of a variable may be specified with `type`:

```
       // Person is a class
     var p : type(Person);
```

The type of `p` is the set of method signatures of class object `Person`. The compiler will create an abstract class `Type$Person` that has the same method signatures as class object `Person`. Then this class will be set as the type of `p`. This mechanism is necessary because at run time there is an object of class `VariableInfo` that describes variable `p` and this object has a method `getType` that returns an object of `ClassInfo`. Since "`type(Person)`" is not a class, it could not be described by this `ClassInfo` object and therefore the compiler replaces "`type(Person)`" by a real class, `Type$Person`.

Apêndice A

The Green Language Grammar

This section describes the language grammar. The reserved words and symbols of the language are shown between " and ". Anything between { and } can be repeated zero or more times and anything between [and] is optional. The prefix Un means the union of two or more rules. The program must be analyzed by unfolding the rule "Program".

There are two kinds of comments:

- anything between `/*` and `*/`. Nested comments are allowed.
- anything after `//` till the end of the line.

Of course, comments are not shown in the grammar.

The rule CharConst is any character between a single quote '. Escape characters are allowed. The rule Str is a string of zero or more characters surrounded by double quotes ". The double quote itself can be put in a string preceeded by the backslash character `\`. The symbol `+` is used to concatenate two strings into a larger one. For example,

```
"ABC" + "DEF"
```

is the same as `"ABCDEF"`. If one of the strings is not literal, the operation is made at run time.

A literal number can have a trailing letter defining its type:

```
35b   // byte number
 2i   // integer number
```

There should be no space between the last digit and the letter.

All words that appear between quotes in the grammar are reserved Green keywords. Besided those, the keyword `subtypeOf` is reserved, which may be used in the future to state a class is *subtype* of another. If a class `Teacher` is declared as subtype of `Person`, the compiler would enforce that `Teacher` defines all methods defined in `Person`.

AddExpr	::=	MultExpr { AddOp MultExpr }
AddOp	::=	"+" | "−"
AndExpr	::=	RelExpr { "and" RelExpr }
ArrayInit	::=	"#" "(" OrExpr { "," OrExpr } ")"
ArrayInitExpr	::=	ArrayInit | Expr
ArrayInitOrExpr	::=	ArrayInit | OrExpr
AssertClause	::=	"assert" ["before" OrExpr ";"]
		{ StatVarDec }
		["after" OrExpr ";"]
		"end"

Assignment ::= PostfixExpr “=” ArrayInitOrExpr
BasicType ::= “boolean” | “byte” | “char” | “double” | “integer” | “long” | “real”
BitExpr ::= ShiftExpr { BitOp ShiftExpr }
BitOp ::= “&” | “|” | “^”
Block ::= “begin” StatementList “end”
BooleanConst ::= “true” | “false”
BreakStat ::= “break”
ByteConst ::= Digit { Digit } “b”
CaseStat ::= “case” Expr “of” EachCase { EachCase }
    [“otherwise” UnStatBlock] “end”
Class ::= Id [“(” TypeList “)”]
ClassType ::= Id [“(” TypeList “)”] | “type” “(” Id “)”
ClassDec ::= ShellClassDec |
    [ClassObjDec] [“abstract”] [“reflective”] “class” Id
    [ClassParamList]
    [“subclassOf” Class]
    UnPubPri
    “end”
ClassObjDec ::= “object” Id [ClassParamList]
    UnObjPubPri
    “end”
ClassParamList ::= “(” ClassParamSpecif { “,” ClassParamSpecif } “)”
ClassParamSpecif ::= Id [“:” ClassType]
ConstExprCase ::= BooleanConst | ByteConst | CharConst | IntegerConst
    | LongConst | Class
ConstDec ::= “const” ConstItem { “,” ConstItem } “;”
ConstExpr ::= BooleanConst | ByteConst | CharConst | DoubleConst
    | IntegerConst | LongConst | RealConst
ConstExprList ::= ConstExprCase { “,” ConstExprCase }
ConstItem ::= Id [“:” ConstType] “=” OrExpr
ConstType ::= BasicType | “String”
Digit ::= “0” | ... | “9”
DoubleConst ::= Digit { Digit } “.” { Digit } [Scale] “d”
E ::= “E” | “e”
EachCase ::= ConstExprList “:” UnStatBlock
EnumDec ::= “enum” “(” Id [“=” OrExp] { “,” Id [“=” OrExp] } “)”
ExceptionClause ::= “(” “exception” : ClassType “)”
Expr ::= PostfixExpr “=” Expr | OrExpr
ExprList ::= Expr { “,” Expr }
FormalParamDec ::= IdList “:” [“...”] Type
FormalParamDecList ::= FormalParamDec { “;” FormalParamDec }
ForStat ::= “for” Id “=” OrExpr “to” OrExpr “do” UnStatBlock
Id ::= Letter { Letter | Digit | “_” }
IdList ::= Id { “,” Id }
IfStat ::= “if” OrExpr “then” StatementList [“else” StatementList] “endif”
InitMethodDec ::= ProcHeading
InitStat ::= PostfixExpr “#” “init” “(” { Expr } “)”
InstVarDec ::= IdList “:” TypeExt “;”

InstVarDecList ::= “var” InstVarDec { InstVarDec }
IntegerConst ::= Digit { Digit } [“i”]
IntegerConstValue ::= IntegerConst | Id
Letter ::= “A” | ... | “Z” | “a” | ... | “z”
LocalDec ::= “var” VarDec { VarDec }
LongConst ::= Digit { Digit } “L”
LoopStat ::= “loop” StatementList “end”
MessageReceiver ::= “super” | “self” | OrExpr | “exception”
MessageSend ::= [MessageReceiver “.”] Id “(” ExprList “)”
MethodDec ::= ProcHeading [AssertClause] [LocalDec] Block
MultExpr ::= BitExpr { MultOp BitExpr }
MultOp ::= “/” | “*” | “%”
ObjPrivatePart ::= ObjVarDecList | MethodDec | ConstDec | EnumDec
ObjPublicPart ::= MethodDec | ConstDec | EnumDec
ObjVarDecList ::= “var” ObjVarDec { ObjVarDec }
ObjVarDec ::= IdList “:” Type [“=” ArrayInitOrExpr] “;”
OrExpr ::= XorExpr { “or” XorExpr }
PostfixExpr ::= PrimaryExpr | PostfixExpr “[” Expr “]” |
MessageReceiver “.” Id [“(” { Expr } “)”]
PrimaryExpr ::= Id | ConstExpr | Str | “(” Expr “)” | “self” | “result” |
“nil” | BasicType
PrivatePart ::= InstVarDecList | MethodDec
ProcHeading ::= [“abstract”] “proc” Id “(” [FormalParamDecList] “)”
[ExceptionClause] [“:” Type]
Program ::= ClassDec { ClassDec }
RealConst ::= Digit { Digit } “.” Digit { Digit } [Scale] [“r”]
Relation ::= “==” | “<” | “>” | “<=” | “>=” | “<>”
RelExpr ::= AddExpr [Relation AddExpr]
RepeatStat ::= “repeat” StatementList “until” Expr
ReturnStat ::= “return” [Expr]
Scale ::= E [“+” | “−”] Digit { Digit }
ShellClassDec ::= “shell” “class” Id “(” ClassType “)” [“subclassOf” Id]
UnPubPri
“end”
ShiftExpr ::= UnaryExpr [ShiftOp UnaryExpr]
ShiftOp ::= “<<” | “>>”
Statement ::= Assignment “;” | MessageSend “;” | “;” | InitStat “;”
| ReturnStat “;” | IfStat | WhileStat | CaseStat | StatVarDec “;”
| ForStat | TryStat | RepeatStat “;” | LoopStat | BreakStat “;”
StatementList ::= { Statement }
StatVarDec ::= “var” Id “:” Type [“=” ArrayInitExpr]
TryStat ::= “try” “(” Expr “)” StatementList “end”
Type ::= BasicType | ClassType | “array” “(” TypeOrExpr “)”
“[” [IntegerConstValue] “]” “[” [IntegerConstValue] “]” | Mes-
sageSend
TypeExt ::= [“@”] Type
TypeList ::= Type { “,” Type }
TypeOrExpr ::= TypeExt | OrExpr

UnaryExpr	::=	PostfixExpr | UnaryOp UnaryExpr
UnaryOp	::=	"~" | "+" | "−" | "not" | "++" | "−−"
UnObjPubPri	::=	[InitMethodDec]
		["public" ":" { ObjPublicPart }]
		["private" ":" { ObjPrivatePart }]
UnPubPri	::=	[InitMethodDec]
		["public" ":" { MethodDec }]
		["subclass" ":" { MethodDec }]
		["private" ":" { PrivatePart }]
UnStatBlock	::=	Statement ";" | "begin" StatementList "end"
VarDec	::=	IdList ":" TypeExt ";"
WhileStat	::=	"while" Expr "do" UnStatBlock
XorExpr	::=	AndExpr { "xor" AndExpr }

Apêndice B

The Introspective Reflection Library

This appendix presents the classes of Green introspective reflection library. The class hierarchy and some built-in Green classes are shown in Figure B.1. Programmers cannot create objects from the classes described in this Chapter.

It would be very expensive to add reflective information to all Green programs since most of them will not need it. Therefore we assume the methods described in this chapter will only work if the programmer has set some compiler options so that the compiler has added reflective information to the object files. If a method is called and the corresponding information is not available, it will throw the unchecked exception

`NoReflectiveInfoException`

Before describing the classes, let us remember some methods of classes `Any` and `AnyClass`. Class `Any` has a method

`getInfo() : AnyObjectInfo`

returns an object that describes the object `self`.

and class `AnyClass` defines methods

To `AnyClass` are added the methods

`getClassInfo() : ClassInfo`

returns an object that describes the class of the object.

`getClassObject() : AnyClassObject`

returns the class object of the object.

Class `AnyValue` defines methods `getInfo`, `getClassInfo`, and `getClassObject` exactly as described above. `AnyValue` is the superclass of the basic classes (`char`, `boolean`, etc).

Class `AnyClassObject` defines methods

`getAssociateClassInfo() : ClassInfo`

```
Any
  AnyClassObject
  AnyClass
    AnyArray
    AnyClassArray

    ClassInfo
        ValueClassInfo
        RefClassInfo
            AbstractClassInfo
            ConcreteClassInfo
                NormalClassInfo
                ArrayClassInfo

    MethodInfo
        ClassMethodInfo
        ClassInitMethodInfo
        ObjectMethodInfo
        ObjectInitMethodInfo



    InstanceVariableInfo
        ClassInstanceVariableInfo
        ObjectInstanceVariableInfo


    AnyObjectInfo
        ObjectInfo
        ClassObjectInfo

    MethodBodyInfo
    AssertionInfo
    VariableInfo
      ParameterInfo
    ConstantInfo
    EnumInfo
    MethodCallInfo
    LiveLocalVariableInfo
      LiveParameterInfo
    CodeAnnotation
```

Figura B.1: Classes of the Introspective Reflection Library returns information about the class associated to this class object.

`getInitMethod() : ObjectInitMethodInfo`
returns object describing the `init` method of the class object (or `nil` if there is no one).

`AnyClassObject` is an abstract class that is *supertype* of all class objects. To every class object the compiler adds all methods defined in `AnyClassObject`. therefore every class object has methods `getInfo`, `getAssociateClassInfo`, and `getInitMethod` for introspective reflection.

The example below show the use of these methods.

```
var p : Person;

p = Person.new();
if p.getClassObject() == Person and
   p.getClassInfo() == Person.getAssociateClassInfo()
then
    // this is executed
  Out.writeln("Ok !");
endif
```

Classes of the Library

Class `ClassInfo`

represents information about a class. Parameterized, shell, and extension classes are not really classes and no `ClassInfo` object describes them at run time. However, these classes are used to create real classes. For example, the declaration

```
var list : DS.List(char);
```

makes the compiler create a class "`List$p$1$char`". The "`p`" between the dollar signals means "parameterized". The "`1`" means there is one parameter. Every class created by the compiler has a name composed by

- a class name which may be a parameterized, shell, or extension class. In this example, this name is "`List`";
- "`$code$`" which indicates the nature of the class or other information. For parameterized classes, "`code`" is "`p`". Shell and extensions use "`sh`" and "`ex`";
- "`$n$`" in which `n` is the number of parameters;
- another class name or information about the class. If more than one name is needed, `$$` is used to separate them. For example, a declaration
  ```
  var dict : DS.Dict(String, Person);
  ```
  causes the creation of a class "`Dict$p$2$String$$Person`".

Note `$` cannot be used in Green identifiers. Therefore the compiler created class names do not conflict with user class names.

There are several ways of implementing parameterized classes. One option reuses the compiled code of the parameterized code for all instantiations of the class. For example, the code for classes `DS.List(Person)`, `DS.List(Figure)`, and `DS.List(Symbol)` would be the same. Another implementation technique would create a new class like `List$p$1$Person` for every instantiation. Whenever the case, there will be one `ClassInfo` object for every instantiated parameterized class. In the previous example, there would be a `ClassInfo` object describing each of the classes `DS.List(Person)`, `DS.List(Figure)`, and `DS.List(Symbol)` even if their code is shared.

If code sharing is used, each method of `List` will be shared among three classes. Suppose variable `mi` refers to a `ClassMethodInfo` object that describes a shared method of `List`. Considering method

```
getDeclaringClass
```

of `ClassMethodInfo` returns a `ClassInfo` object describing the class in which the method `m` is, what a call

```
mi.getDeclaringClass()
```

would return ? The answer is: `ClassMethodInfo` has no `getDeclaringClass` method. This illustrates one more example of the usefulness of designing the whole Green language at once. We had time to modify everything before releasing the language. We had opportunity to make all the parts work together without unwanted side effects.

Methods:

`getName() : String`
returns the name of the class.

`toString() : String`
returns the name of the class. This method may be modified by the user to return more data about the class.

`isSupertypeOf( aSubtype : ClassInfo ) : boolean`
returns `true` if the class described by `aSubtype` is *subtype* of the class described by `self`.
Example:

```
var w : Window = Window.new();
  // TextWindow is subtype of Window
var tw : TextWindow = TextWindow.new();

if w.getClassInfo().isSupertypeOf( tw.getClassInfo() )
then
  // always executed
endif
```

`isSuperclassOf( aSubclass : ClassInfo ) : boolean`
returns `true` if the class described by `self` is *superclass* of the class described by `aSubclass`.

`getSuperclass() : ClassInfo`
returns object describing the superclass. Returns `nil` if there is no superclass; that is, the class described by `self` is `AnyValue` or `Any`.

`getInstanceVariables() : DS.Iter(ClassInstanceVariableInfo)` returns an iterator with all instance variables of the class including the inherited ones. The first elements are the instance variables of the class described by `self`, then those of the superclass and so on. There may be two variables of different classes with the same name.

```
getMethods() : DS.Iter(ClassMethodInfo)
```

returns an iterator with objects describing the class methods. It includes all the inherited methods from sections `public`, `subclass`, and `private`. It does not include the constructors (`init` methods) which are considered special methods. Note these constructors are not the `new` methods of the class objects. They do not create objects of the class.

```
getPublicMethods() : DS.Iter(ClassMethodInfo)
```

returns an iterator with objects describing the public class methods. This includes inherited methods but not methods overridden in subclasses. That is, if the receiver of message `getPublicMethods` represents a class `B` that overrides method `m` inherited from its superclass `A`, the iterator will return only `B::m`.

```
getInitMethods() : DS.Iter(ClassInitMethodInfo)
```

returns an iterator with objects describing the class constructors, the `init` methods. This is the only method of `ClassInfo` that returns information on constructors.

```
getInstanceVariable(name : String) : ClassInstanceVariableInfo
```

returns object describing the instance variable whose name is passed as parameter. The search is made in this class, then in the superclass, and so on. The first instance variable found is returned. Remember a class may have an instance variable with name equal to another instance variable of its superclass. If there is no variable with this name, `nil` is returned.

```
getMethod( name : String;
           paramTypes : array(ClassInfo)[] ) : DS.Iter(ClassMethodInfo)
```

returns an iterator with objects describing the methods with names `name` whose parameter types are in array `paramTypes`. The search is made in this class, its superclass, and so on. If there is no method with this name and parameter types, `nil` is returned. If `paramTypes` is `nil`, `getMethod` assumes method `name` has no parameter. `init` methods (constructors) are not considered by this method.

```
getMethod_v( name : String;
             paramTypes : ... array(AnyClassObject)[] ) : DS.Iter(ClassMethodInfo)
```

Similar to method `getMethod` except that here `paramTypes` represents a variable number of parameters. Each parameter should be a class object:

```
var acc : Account;
var  ai : ClassInfo;
...
acc.deposit(100.0);
ai = acc.getClassInfo();
```

```
var mi : ClassMethodInfo;
mi = ai.getMethod_v( "deposit", real );
Out.writeln( mi.getName() );   // prints "deposit"
```

Again, constructors (`init` methods) are not taken into account.

`getPublicMethod( name : String ) : DS.Iter(ClassMethodInfo)`
returns an iterator with all public methods named `name`. Inherited methods are included if they were not overridden in subclasses. That is, if `self` represents class `B` that overrides method `m` inherited from `A`, only `B::m` will be returned by the iterator. `init` methods are not taken into account.

`getThisClassInstanceVariables() : DS.Iter(ClassInstanceVariableInfo)`
returns an iterator with objects describing all the instance variables declared in this class — inherited ones are not considered.

`getThisClassMethods() : DS.Iter(ClassMethodInfo)`
returns an iterator with objects describing the methods declared in this class. Neither inherited methods nor `init` methods are taken into account.

`getThisClassPublicMethods() : DS.Iter(ClassMethodInfo)`
returns an iterator with objects describing the public methods declared in this class. Inherited methods are not taken in account.

`getAssociateClassObject() : AnyClassObject`
returns the class object corresponding to the class. So the expression
`p.getClassInfo().getAssociateClassObject() == Person`
will be `true` if object `p` points to an object of class `Person`.

`isClassOf( any : Any ) : boolean`
returns `true` if `any` is an object of the class described by `self` (the object that received this message). Note this method returns `false` if the class of `any` is a subclass or subtype of the class described by `self`. The expression
`p.getClassInfo().isClassOf(p)`
is always `true`.

`isAbstract() : boolean`
returns `true` if the class is abstract.

`isReflective() : boolean`
returns `true` if the class if reflective. See Chapter 12 for the definition of reflective classes.

`getNotes() : DS.Iter(CodeAnnotation)`
returns an iterator with annotations on the class. See description of `CodeAnnotation`.

Class `ValueClassInfo` subclass of `ClassInfo`

represents information about a basic class as `byte` or `char`. These classes inherit from `AnyValue` which does not inherit from anyone. Basic classes do not have instance variables[1] or constructors. The operators are considered methods. So, one could write

```
    var ci : ClassInfo;
    ci = 5.getClassInfo();
    Out.writeln( ci.getMethod("+").next().
        getParameters().next().getType().getName() );
```

to write the name of the first parameter type ("`integer`") of method "`+`" of the class of "`5`", which is "`integer`" too.

Methods:
none. The goal of this class is allow one to discover if a `ClassInfo` object describes a value class. This is made by a cast:

```
var vi : ValueClassInfo;
var ci : ClassInfo;
...
try(catch)
  vi = ValueClassInfo.cast(ci);
  ...
end
```

If the cast succeeds, `ci` pointed to a `ValueClassInfo` object.

Class `RefClassInfo` subclass of `ClassInfo`

represents information about a reference class; that is, any class that is not a built-in class like `char` or `integer`.

Methods: none.

Class `AbstractClassInfo` subclass of `RefClassInfo`

represents information about an abstract class.

Methods: none.

Class `ConcreteClassInfo` subclass of `RefClassInfo`

represents information about a non-abstract class.

Methods: none.

Class `NormalClassInfo` subclass of `ConcreteClassInfo`

[1] Or at least the reflective library thinks so. The implementation *could* be different !

represents information about a non-abstract and non-array class.

Methods: none.

Class `ArrayClassInfo` subclass of `ConcreteClassInfo`

represents information about array classes.

Methods:

`getArrayElementClass() : ClassInfo`
returns object describing the class of the array elements.

`getNumberOfDimensions() : integer`
returns the number of array dimensions.

Class `AnyArray` subclass of `Any`

this class is superclass of all array classes.

Methods:

```
set( v : Any; i : integer )
   ( exception : CatchAnyArrayException )
```

set to `v` the `i`th position of the array. The `exception` variable is used to throw exceptions, described in Chapter 11. The methods that follow work similarly with arrays of two or more dimensions. These methods may throw the following exceptions:

- `TypeErrorException`. This exception is thrown if there is a type error; that is, if the array element type is `T` and the run-time type of object `v` is not a subtype of `T`;
- `TooManyDimensionsException`, which is thrown if the array has less dimensions than those specified. For example, if one uses the next method `set` with a one-dimensional array. There should have been specified only only index, `i` or `j`.

```
set( v : Any; i, j : integer )
   ( exception : CatchAnyArrayException )
```

```
set( v : Any; i, j, k : integer; others : ... array(integer)[] )
   ( exception : CatchAnyArrayException )
```

```
get( i : integer ) : Any
   ( exception : CatchAnyArrayException )
```

returns the element of the array index `i`. The methods that follow work similarly with arrays of two or more dimensions. The `get` methods may throw exception `TooManyDimensionsException`.

```
get( i, j : integer ) : Any
```

`( exception : CatchAnyArrayException )`

`get( i, j, k : integer, others : ... array(integer)[] )`
`( exception : CatchAnyArrayException )`

`getSize() : integer`
returns the array size.

`toString() : String`
returns a string like "`array(T)[][]`" if the array has two dimensions and `T` as the element type.

Class `MethodInfo`

represents information about methods.

Methods:

`getName() : String`
returns the method name.

`toString() : String`
returns a string with the method name and the parameter names and types. If mi represents a method

```
proc get( i, j : integer ) : Person
```

then "`mi.toString()`" would return the string
"`get( i : integer; j : integer ) : Person`"

`getBodyInfo() : MethodBodyInfo`
`( exception : CatchNoReflectiveBodyInfoException )`
returns object describing the method local variables and statements. Throws exception `NoReflectiveBodyInfoException` if no reflective information about the method body is available.

`getVisibility() : integer`
return an integer `MethodInfo.constructor_v`, `MethodInfo.public_v`, `MethodInfo.subclass_v`, or `MethodInfo.private_v` telling the method visibility.

`getParameterTypes() : DS.Iter(ClassInfo)`
returns an iterator with all the parameter types.

`getReturnType() : ClassInfo`
returns object describing the method return type. If there is no return type, nil is returned.

`getParameters() : DS.Iter(ParameterInfo)`
returns an iterator with objects describing the method parameters in the order they were

declared.

`getExceptionClass() : ClassInfo`
returns object describing the class of the parameter `exception` of the method. If there is no `exception` parameter, `nil` is returned.

`getNotes() : DS.Iter(CodeAnnotation)`
returns an iterator with annotations on the method.

Class `ClassMethodInfo` subclass of `MethodInfo`

represents a method of a class. The class in which the method is cannot be discovered using the methods of `ClassMethodInfo`, including the inherited ones from `MethodInfo`. This was made on purpose to allow software tools (or the compiler itself) to plug a method to more than one class.

Methods:

`isAbstract() : boolean`
returns `true` if this method is abstract.

```
invoke( obj : Any; args : array(Any)[] ) : Any
      ( exception : CatchInvokePackedException )
```
if `self` describes a method `m`, `invoke` would be equivalent to
```
obj.m( args[0], args[1], ... args[n] )
```
in which `n` is the index of the last `args` object. This method will throw exception `PackedException` if method `m` has thrown an exception. The exception thrown by `m` is packed in an exception object of class `PackedException`. If the arguments cannot be *cast* to the method arguments, or the number of arguments is wrong, an exception `WrongParametersException` is thrown. If `obj` does not have the method described by `self`, exception `TypeErrorException` is thrown.

Note method `invoke` may be used to call a method of `public`, `subclass`, or `private` sections. It will never call constructor since constructors are described by class `ClassInitMethodInfo`.

```
invoke_v( obj : Any; args : ... array(Any)[] ) : Any
      ( exception : CatchInvokePackedException )
```
the same as `invoke` but with a variable number of parameters.

Class `ClassInitMethodInfo` subclass of `MethodInfo`

represents an `init` method of a class, a constructor. A constructor cannot be called by the user outside its class. So, there is no `invoke` method here.

Methods: none.

Class `ObjectMethodInfo` subclass of `MethodInfo`

represents a method of an object which may have a class or may be a class object.

Methods:

`invoke( args : array(Any)[] ) : Any`
`( exception : CatchInvokePackedException )`

if `self` describes a method `m` of object `obj`, `invoke` is equivalent to call `m` on `obj` using `args` as parameters. This method will throw exception `PackedException` if method `m` has thrown an exception. The exception thrown by `m` is packed in an exception object of class `PackedException`.

If the arguments cannot be cast to the method arguments or the number of arguments is wrong, exception `WrongParametersException` is thrown.

Then the following situation may occur: class `A` defines method `m()` which is overridden in a subclass `B` of `A`. Object `objmi` describes method `A::m` but `objmi` belongs to class `B`. Then

```
objmi.invoke(args)
```

will call `A::m`. Objects like `objmi` will never be returned by method `getMethods` of `AnyObjectInfo` (see ahead) which would return object describing `B::m` since `objmi` is linked to a `B` object. However, the shell method `interceptAll` uses objects like `objmi` in which the *method* and the *object* described do not match.

`invoke_v( args : ...array(Any)[] ) : Any`
`( exception : CatchInvokePackedException )`

the same as the previous `invoke` but with a variable number of parameters.

Class `ObjectInitMethodInfo` subclass of `MethodInfo`

represents an `init` method of a class object. A class-object constructor cannot be called outside the class object. So, there is not `invoke` method here.

Methods: none.

Class `InstanceVariableInfo`

describes an instance variable.

Methods:

`getName() : String`

returns the name of the instance variable.

`getType() : ClassInfo`

returns the instance variable type.

`isExpanded() : boolean`

returns `true` if the variable is expanded. That is, the variable is declared with `@` and obeys value semantics.

`getNotes() : DS.Iter(CodeAnnotation)`

returns an iterator with annotations on the instance variable.

Class `ClassInstanceVariableInfo` subclass of `InstanceVariableInfo`

describes an instance variable of a class.

Methods:

`set( obj : Any; v : Any )`
`( exception : CatchTypeErrorException )`

set the variable described by `self` of object `obj` to `v`. Exception `TypeErrorException` is thrown if there is a type error; that is, if the instance variable type is `T` and the run-time type of object `v` is not a subtype of `T`. Or if `obj` does not have the instance variable described by `self`.

`get( obj : Any ) : Any`
`( exception : CatchTypeErrorException )`

return the value of the variable described by `self` of the object `obj`. Exception `TypeErrorException` is thrown if the object `obj` does not have the instance variable described by `self`.

Class `ObjectInstanceVariableInfo` subclass of `InstanceVariableInfo`

describes a variable of an object.

Methods:

`set( v : Any )`
`( exception : CatchTypeErrorException )`

sets the variable to `v`. Exception `TypeErrorException` is thrown if there is a type error; that is, if the variable type is `T` and the run-time type of object `v` is not a subtype of `T`.

`get() : Any`

returns the variable value.

Class `AnyObjectInfo`

describes an object.

Methods:

`getObject() : Any`

returns the object that `self` describes.

`getInstanceVariables() : DS.Iter(ObjectInstanceVariableInfo)[]`

returns an iterator with objects describing all variables of the object described by `self`.

`getMethods() : DS.Iter(ObjectMethodInfo)`
returns an iterator with objects describing all methods of the object described by `self`. If the object has a class, this includes all inherited methods from `public`, `subclass`, and `private` sections. This does not include the constructors.

If the object does not have a class (it is a class object), the iterator will return objects for all public and private methods of the class object. This excludes any methods `init` of the class object.

`getPublicMethods() : DS.Iter(ObjectMethodInfo)`
returns an iterator with objects describing the public object methods. If the object has a class, this includes inherited methods but not overridden methods. That is, if `self` represents an object of class `B` that overrides method `m` inherited from superclass `A`, the iterator will return only `B::m`.

If the object is a class object, the iterator yields all public methods of the class object.

`getPublicMethod( name : String ) : DS.Iter(ObjectMethodInfo)`
returns an iterator with all public object methods named `name`. If the object has a class, inherited methods are included. But only if they were not overridden in subclasses. That is, if `self` represents an object of class `B` that overrides method `m` inherited from superclass `A`, the iterator will return only `B::m`. Constructors are not taken into account.

`getMethod( name : String; paramTypes : array(ClassInfo)[] ) : ObjectMethodInfo`
returns object describing the object method `name` whose parameter types are in array `paramTypes`. If the object has a class, the search for the method is made in this class, its superclass, and so on. Public, subclass, and private methods are considered.

If the object is a class object, the search is made in the public and private sections of the object. The constructor `init()` is not taken into account.

If not method is found, `nil` is returned. If `paramTypes` is `nil`, `getMethod` assumes the method to be searched has no parameters.

`getMethod( name : String ) : ObjectMethodInfo`
returns object describing method `name` of `self`. If there is not any method with this name, `nil` is returned. The normal method look-up is made to find the method. The first method found is returned.

`getInstanceVariable( name : String ) : ObjectInstanceVariableInfo`
returns object describing variable called `name`. It is returned the first variable found in a search from the object class to its superclass, if the object has a class. If there is no variable with this name, `nil` is returned.

`getTypeInfo() : ClassInfo`
returns object describing a class that has the same type as the object. The type of an object has the same definition as the type of a class. This method returns an object describing

- the object class if the object has a class;
- an abstract class if the object is a class object. This abstract class has the same type as the object.

Class `ObjectInfo` subclass of `AnyObjectInfo`

describes an object that has a class

Methods:

No methods in this class. Used only to discriminate objects that were created from a class from class objects that are the classes.

Class `ClassObjectInfo` subclass of `AnyObjectInfo`

describes a class object.

Methods:

`getInitMethod() : ObjectInitMethodInfo`

returns an object describing the `init` method of the class object, if one exists.

`getPublicConstants() : DS.Iter(ConstantInfo)`

returns an iterator with objects describing the public constants of the class object, which does not include enumerated constants.

`getConstants() : DS.Iter(ConstantInfo)`

returns an iterator with objects describing all constants of the class object, which does not include enumerated constants.

`getPublicEnumConstants() : DS.Iter(EnumInfo)`

returns an iterator with objects describing the public enumerate declarations of the class object.

`getEnumConstants() : DS.Iter(EnumInfo)`

returns an iterator with objects describing all enumerate declarations of the class object.

`getNotes() : DS.Iter(CodeAnnotation)`

returns an iterator with annotations on the class object.

```
new( args : array(Any)[] )
    ( exception : CatchNewException ) : Any
```

if `coi` describes class `Person`, a call `coi.new(args)` is equivalent to

```
Person.new( args[0], args[1], ... args[n] )
```

in which `n` is the last valid index of `args`. The first new method that accepts the parameters is called. This method will throw exception `PackedException` if the class method `new` has thrown an exception. The exception is packed in an exception object of class `PackedException`.

Exception `WrongParametersException` will be thrown if no `new` method of the class object matches the parameters passed to the above method `new`. That is, the types or number of parameters is wrong.

Exception OutOfMemoryException will be thrown if there is not sufficient memory for creating the object.

```
new_v( args : ... array(Any)[] )
     ( exception : CatchNewException ) : Any
```

the same as the previous method new but with variable number of parameters.

Class MethodBodyInfo

describes the assertions, local variables, and the statements of a method.

Methods:

```
getMethodInfo() : MethodInfo
```

returns object describing the method.

```
getAssertionInfo() : AssertionInfo
```

returns object describing the method assertions. If there is no assertion, nil is returned.

```
getLocalVariables() : DS.Iter(VariableInfo)
```

returns an iterator with objects describing the method local variables. This includes all local variables, including those declared inside the begin-end block.

It is worth noting there would be problems if a block inside the begin-end of a method introduced a new scope. Then a method

```
proc search() : boolean
  var i : integer;
  begin
  ...
  if i > 0
  then
    var i : integer;
    ...
  endif
  end
```

would have two different variables with the same name. At least confusing. This example enforces what we have said: it is better to design the whole language at once so incompatibilities among the language elements can be elegantly avoided.

```
getStatements() : DS.Iter(StatementNo)
```

returns an iterator with objects describing the method statements in the order they appear in the method declaration. StatementNo and other classes related to the Abstract Syntax Tree of the compiler will not be described in this report.

Class AssertionInfo

describes the assertion section of a method. None of the following methods reveals the method in which the assertion is.

`getBeforeExpression() : ExprAST`
returns object describing the expression following keyword `before` in the assertion clause. Class `ExprAST` belongs to the Abstract Syntax Tree of the compiler and is not described in this report. If there is no `before` expression, `nil` is returned.

`getAfterExpression() : ExprAST`
returns object describing the expression following keyword `after` in the assertion clause. Class `ExprAST` belongs to the Abstract Syntax Tree of the compiler and is not described in this report. If there is no `after` expression, `nil` is returned.

`getStatVarDeclarations() : DS.Iter(StatVarDecNo)`
returns an iterator with objects describing the initialization of variables after the `before` expression. As an example, if the assertion clause is

```
assertion
  before not full();
  var oldSize : integer = getSize();
  after oldSize == getSize() - 1;
end
```

This method would return a one-element iterator with an object describing variable `oldSize` and the expression that is assigned to it, "`getSize()`".

Class `VariableInfo`

describes a local variable.

Methods:

`getName() : String`
returns the variable name.

`getType() : ClassInfo`
returns the variable type.

`getDeclaringMethod() : MethodInfo`

returns object describing the method in which this local variable was declared.

`isExpanded() : boolean`
returns `true` if the variable is expanded. That is, the variable is declared with `@` and obeys value semantics. Note parameters are never expanded.

`getNotes() : DS.Iter(CodeAnnotation)`

returns an iterator with annotations on the variable.

Class `ParameterInfo` subclass of `VariableInfo`

describes a method parameter.

Methods:

`setDeclaringMethod( MethodInfo declaringMethod )`
sets the method declaring this parameter.

`isVariableNumber() : boolean`
returns `true` if this parameter is an array used to make the method accept a variable number of parameters, as `v` in

```
proc print( f : Font; v : ... array(Any)[] )
```

Then if `pi` describes parameter `v` of this method,

```
pi.isVariableNumber()
```

would return `true`.

Class `ConstantInfo`

describes a constant.

Methods:

`getName() : String`
returns the variable name.

`getType() : ClassInfo`
returns the variable type.

`getVisibility() : integer`
return an integer `MethodInfo.public_v`, or `MethodInfo.private_v` telling the constant visibility.

`getValue() : Any`
returns the constant value.

`getNotes() : DS.Iter(CodeAnnotation)`
returns an iterator with annotations on the constant.

Class `EnumInfo`

describes a declaration of enumerated constants.

Methods:

`getConstants() : DS.Iter(ConstantInfo)`
returns objects describing each of the enumerated constants.

`getVisibility() : integer`
return an integer `MethodInfo.public_v`, or `MethodInfo.private_v` telling the enumeration visibility.

`getNotes() : DS.Iter(CodeAnnotation)`
returns an iterator with annotations on the enumerate.

To object `Runtime` are added the following methods:

`getClasses() : DS.Iter(ClassInfo)`
returns an iterator with objects describing all program classes. Note shell, extension, and parameterized classes are not real classes and no object describes them. However, the compiler creates some classes based on these classes that are returned by this method and can be searched using the following method.

`searchForClass( name : String ) : ClassInfo`
returns object describing class `name`. The search is made among all program classes. If no class is found, `nil` is returned.

```
getMethodCallStack() : DS.Stack(MethodCallInfo)
                  ( exception : CatchNoReflectiveCallInfoException )
```
returns a stack with one object for each method in the method call stack. There will always be at least one method in the stack. Exception `NoReflectiveCallInfoException` is thrown if the program was not compiled with information about the run-time stack.

`getCatchObjectStack() : DS.Stack(Catch)`
returns a stack with catch objects.

The following code prints the name of all exceptions that can be caught by the active catch objects, which are those returned by `getCatchObjectStack`. This method will always return a correct stack at run time, even if the program was not compiled with reflective information. An example of use of this method is given below.

```
var cmi : ClassMethodInfo;
var iterMethod : DS.Iter(ClassMethodInfo);
var iterStack  : DS.Iter(Catch) =
Runtime.getCatchObjectStack().getIter();

while iterStack.more() do
  begin
  iterMethod = iterStack.next().getClassInfo().getPublicMethods();
  while (cmi = iterMethod.next()) <> nil do
```

```
    if cmi.getName().equals("throw")
    then
      Out.writeln( cmi.getParameters().next().getType().getName() );
    endif
  end
```

Note we used two different techniques to scan the elements of the iterators.

It is not possible to add or remove catch objects from the stack. But one may attach a shell to a catch object of the stack. Thus a `throw` method of a shell may be called when an exception is thrown.

`getMethodThrownException()`
`(exception : CatchNoExceptionInfoException) : MethodInfo`

return information on the method that has thrown the last exception. Exception
`NoExceptionInfoException`
is thrown if the program was not compiled to keep the information requested by this method.

`setCatchUnchecked( myCatch : CatchUncheckedException )`

set the first reference of the stack of catch objects to `myCatch`. Since the type of this object is `CatchUncheckedException`, it is able to treat all unchecked exceptions. This works as follows. Before the program starts, a default catch object is pushed into th stack of catch objects. This object will be used to treat the unchecked exceptions of th program (unless the program catches them itself). The `setCatchUnchecked` method just allow the user to change this default catch object.

`getCatchUnchecked() : CatchUncheckedException`

get the first object of the stack of catch objects.

Class `MethodCallInfo`

describes a method call. The method is in the method call stack and therefore its local variables and parameters are live.

Methods:

`getMethodInfo() : MethodInfo`

returns object describing the method.

`getLiveLocalVariables() : DS.Iter(LiveLocalVariableInfo)`

returns an iterator with objects describing the local variables of the method, which are, of course, live.

`getLiveParameters() : DS.Iter(LiveParameterInfo)`

returns an iterator with objects describing the method parameters, which are, of course, live.

Class `LiveLocalVariableInfo`

describes a local variable of a method that is currently active; that is, the method is in the stack of called methods.

Methods:

`getVariableInfo() : VariableInfo`
returns information about the variable.

`set( v : Any )`
`( exception : CatchTypeErrorException )`
sets the variable to `v`. If the run time type of `v` is not a subtype of the type of the variable, exception `TypeErrorException` is thrown.

`get() : Any`
returns the variable value.

Class `LiveParameterInfo` subclass of `LiveLocalVariableInfo`

Methods: none.

Abstract Class `CodeAnnotation`

an annotation about a class, method, or variable. The object returned by method `getDescription` of this class should be *cast* to some useful type. An annotation gives further information which is not related to the structure of the class, method, or variable. For example, an object of `CodeAnnotation` attached to a method could tell if the method changes the object instance variables or calls other object methods. If it does not, a debugger could allow the programmer to call this read-only method in an Inspect window. This information is available to the compiler and given to the programmer through a `CodeAnnotation` object.

A *final* class cannot be subclassed and has no *subtype*. A *final* method cannot be overridden in a subclass. Green has no `final` keyword to mark final classes and methods. These classes and methods should be marked final by the programming environment or through a special statement in the module system.[2] There should be a "final" code annotation to each final class or method.

Objects of `CodeAnnotation` can provide other information that would only be available in the program documentation. For example, a `CodeAnnotation` object linked to a class `Button` could tell which methods are used to add and remove a listener to the button. A button is a graphical component that may be pressed with the mouse. When this occurs, a message "`actionPerformed`"[3] is sent to each of the button listeners which are objects with a `actionPerformed` method.

Figures B.2 and B.3 show examples of use of `CodeAnnotation` objects. In Figure B.2 a test is made to discover if a method changes the object instance variables or call other methods. The second example in Figure B.3 prints the names of methods of a class that are used to add and remove listeners. It uses a class `EventSourceInfo` which keeps information on the methods.

We hope the compile-time objects created by the programmer will be able to add `CodeAnnotation` objects to classes, methods, and variables as the compiler itself.

[2] Yet to be designed.
[3] We are using the names employed in Java AWT.

```
var change : boolean;
var an     : CodeAnnotation;
var mi     : ObjectMethodInfo;
var iter   : DS.Iter(CodeAnnotation);


  //  mi will hold data on method get of object x
mi = x.getInfo().getMethod("get", nil);
iter = mi.getNotes();

if (an = iter.next()) <> nil  and
   an.getName().equals("changeState")
then
    // the name of the annotation is changeState
  try(HCatchAll)
    change = boolean.cast(an.getDescription());
    if not change
    then
      Out.writeln("Read only method");
    endif
  end
endif
```

Figura B.2: Test to discover if a method is read-only

```
class EventSourceInfo
  public:
      // only the method headers are shown
    proc getAddMethod() : ClassMethodInfo
    proc getRemoveMethod() : ClassMethodInfo
    ...
end

...

var an     : CodeAnnotation;
var aClass : AnyClassObject;
var esi    : EventSourceInfo
var ci     : ClassInfo;
var iter   : DS.Iter(CodeAnnotation);

ci = aClass.getAssociateClassInfo();
iter = ci.getNotes();

if (an = iter.next()) <> nil   and
   an.getName().equals("eventSource")
then
  try(HCatchAll)
    esi = EventSourceInfo.cast( an.getDescription() );
  end
  Out.writeln( "add method : "    , esi.getAddMethod().getName(), "\n",
               "remove method : ", esi.getRemoveMethod().getName() );
endif
```

Figura B.3: Prints the names of the methods to add and remove listeners

Methods:

`getDescription() : Any`
returns an object describing the annotation.

`getName() : String`
returns the annotation name.

Apêndice C

The Exception Library

The predefined Green exceptions are defined below. The class hierarchy is shown in Figure C.1. All exception classes redefine method `toString` to return a message explaining the exception.

Class `Exception`

an abstract class that is superclass of all predefined exception classes.

Methods: none.

Class `TypeErrorException` subclass of `Exception`

exception thrown by methods `set` of class `AnyArray`, `ClassInstanceVariableInfo`, `ObjectInstanceVariableInfo`, and `LiveLocalVariableInfo` if there is a type error.

Also thrown if a cast to the type of a class failed. Then, if the cast

```
rectangle = Rectangle.cast(figure);
```

fails, exception `TypeErrorException` will be thrown.

Constructors:

`init()`

Methods: none.

Class `WrongParametersException` subclass of `Exception`

this exception will be thrown by method `new` of `ClassObjectInfo` if no `new` method of the class object matches the parameters passed to the above method `new`. Other methods like `invoke` of

```
Exception
  TypeErrorException
  WrongParametersException
  NotFoundException
  PackedException
  TooManyDimensionsException

  MetaException
    ClassNotInAllowedSetException
    NoShellException
    NoExtensionException

  UncheckedException
    StackOverflowException
    IllegalArrayIndexException
    OutOfMemoryException
    InternalErrorException
    MessageSendToNilException

    NoReflectiveInfoException
      NoReflectiveBodyInfoException
      NoReflectiveCallInfoException

    ArithmeticException
      DivisionByZeroException
      RealOverflowException
      RealUnderflowException

    AssertionException
      AssertionAfterException
      AssertionBeforeException
      AssertionCastCharException
      AssertionCastBooleanException
      AssertionCastByteException
      AssertionCastIntegerException
      AssertionCastLongException
      AssertionCastRealException
      AssertionCastDoubleException
```

Figura C.1: Hierarchy of predefined Green exceptions

`ClassMethodInfo` may throw this exception for similar reasons.

Constructors:

`init()`

Methods: none

Class `NotFoundException` subclass of `Exception`

usually this exception is thrown by search methods when the item searched is not found.

Constructors:

`init()`

Methods: none

Class `PackedException` subclass of `Exception`

thrown by methods `invoke` of classes `ObjectMethodInfo` and `ClassMethodInfo` when the executed method throws an exception that is packed by an object of `PackedException`.

Constructors:

`init( exc : Exception )`

packs the exception passed as parameter.

Methods:

`getException() : Exception`

returns the exception packed by the receiver object.

Class `TooManyDimensionsException` subclass of `Exception`

used to signal that an array has less dimensions than those specified in the parameters of method `set` of `AnyArray`.

Constructors:

`init( givenNumber, numDimensions : integer )`

the parameters specify the number of parameters given in the method and the array dimension.

Methods:

`getArrayDimension() : integer`
returns the number of array dimensions.

`getGivenNumber() : integer`
returns the number of elements used to index the array.

Class `UncheckedException` subclass of `Exception`

this abstract class is superclass of all unchecked Green exceptions. These exceptions need not to be caught by the program when thrown by the run-time system or compile-created code.

Constructors: none.

Methods: none.

Class `StackOverflowException` subclass of `UncheckedException`

thrown when there is no more space for the run-time call stack.

Constructors:

`init()`

Methods: none.

Class `IllegalArrayIndexException` subclass of `UncheckedException`

thrown when an illegal index is used in an array.

Constructors:

`init( index : integer; theArray : AnyArray )`
`index` was used to index array `theArray`.

Methods:

`getIndex() : integer`
returns the index.

`getArray() : AnyArray`
returns the array.

Class `OutOfMemoryException` subclass of `UncheckedException`

thrown when there is no more dynamic memory.

Constructors:

`init()`

Methods: none.

Class `InternalErrorException` subclass of `UncheckedException`

thrown when there is an internal error in the run-time system.

Constructors:

`init( s : String )`

`s` is a description on the error.

Methods:

`getErrorString() : String`

returns the error string passed as parameter to the constructor.

Class `MessageSendToNilException` subclass of `UncheckedException`

thrown when a message is sent to `nil`.

Constructors:

`init()`

Methods: none.

Class `NoReflectiveInfoException` subclass of `UncheckedException`

abstract class used as a superclass to the exception classes used to signal errors in the reflective system of Green.

Constructors: none.

Methods: none.

Class `NoReflectiveBodyInfoException` subclass of `NoReflectiveInfoException`

thrown if there is no reflective information about the body of a method.

Constructors:

`init()`

Methods: none.

Class `NoReflectiveCallInfoException` subclass of `NoReflectiveInfoException`

exception used to signal that there is no reflective information about the run-time method call stack.

Constructors:

`init()`

Methods: none.

Class `ArithmeticException` subclass of `UncheckedException`

abstract class of all arithmetic exception classes.

Constructors:

`init()`

Methods: none.

Class `DivisionByZeroException` subclass of `ArithmeticException`

Constructors:

`init()`

Methods: none

Class `RealOverflowException` subclass of `ArithmeticException`

thrown by the run-time system in overflow of real numbers.

Constructors:

`init()`

Methods: none.

Class `RealUnderflowException` subclass of `ArithmeticException`

thrown by the run-time system in underflow of real numbers.

Constructors:

`init()`

Methods: none.

Class `AssertionException` subclass of `1UncheckedException`

abstract superclass of the exception classes for assertions.

Constructors: none

Methods:

`getMethodInfo() : MethodInfo`
returns an object describing the method in which the assertion is.

Class `AssertionBeforeException` subclass of `AssertionException`

Constructors:

`init( mi : MethodInfo )`
the `before` part of the assertion clause of the method described by `mi` evaluated to `false`.

Methods: none.

Class `AssertionAfterException` subclass of `AssertionException`

Constructors:

`init( mi : MethodInfo )`
the `after` part of the assertion clause of the method described by `mi` evaluated to `false`.

Methods: none.

the classes that follow are used in casts among the basic classes. All of them are subclasses of `AssertionException`. An object of, say, `AssertionCastCharException` is thrown if a cast from something to `char` failed. The compiler should have some option to disable the assertions of methods `cast` of basic classes.

```
AssertionCastCharException
AssertionCastBooleanException
AssertionCastByteException
AssertionCastIntegerException
AssertionCastLongException
AssertionCastRealException
AssertionCastDoubleException
```

All of these classes are similar. As an example we describe class `AssertionCastCharException`.

```
class AssertionCastCharException subclassOf AssertionException
    proc init( p_originalValueClass : AnyClassObject; p_value : Any )
      begin
      originalValueClass = p_originalValueClass;
      value = p_value;
      end
  public:
    proc getOriginalValueClass() : AnyClassObject
      begin
      return originalValueClass;
      end
    proc getOriginalValue() : Any
      begin
      return value;
      end
  private:
    var originalValueClass : AnyClassObject;
        value : Any;
end
```

`originalValueClass` is the class of the object that could not have been converted to `char`. The value that could not have been converted to `char` was transformed into an object of a wrapper class which is `p_value`.

Class `MetaException` subclass of `Exception`

abstract superclass of all classes related to the meta level.

Constructors: none.

Methods: none.

Class `ClassNotInAllowedSetException` subclass of `MetaException`

an object of this class is thrown if the program tries to attach a shell to an object of a class not prepared for being used with shells. That is, the class is not in the "allowed set" of the shell class. See Chapter 12 for more information.

Constructors:

`init( value : Any )`
`value` is the object one tried to attach a shell.

Methods:

`getValue() : Any`
returns the object one tried to attach to a shell.

Class `NoShellException` subclass of `MetaException`

an object of this class is thrown if the program tries to remove a shell from an object without a shell.

Constructors:

`init( any : Any )`
`any` is the object from which the program tried to remove the shell.

Methods:

`getObject() : Any`
returns the object from which the program tried to remove a shell.

Class `NoExtensionException`

an object of this class is thrown if the program tries to remove an extension from a class without an attached extension.

Constructors:

`init( aClass : AnyClassObject )`
`aClass` is the class from which the program tried to remove the extension.

Methods:

`getTheClassObject() : AnyClassObject`
returns the class object from which the program tried to remove an extension.

Apêndice D

The Main Green Classes

This Chapter presents the main Green classes and objects along their methods. Figure D.1 shows a hierarchy of Green classes. In this figure are also shown the array class "`array(char)[]`" and some class objects like `Runtime` and `Out`. These objects are preceded by `*` to indicate they are not classes. Each class object is *subtype* of `AnyClassObject`. The container and stream hierarchies are shown in Figure D.2. The hierarchy of the Introspective Reflection Library is shown in Figure B.1.

This appendix have not been updated with the last modifications in this report. Refer to the report itself to get precise information.

```
AnyValue
  char
  boolean
  byte
  integer
  long
  real
  double
Any
  AnyClassObject
    *In
    *Out
    *OutError
    *Screen
    *Storage
    *Runtime
    *Memory
    *char
    *boolean
    *byte
    *integer
    *long
    *real
    *double
  AnyClass
    Char
    Boolean
    Byte
    Integer
    Long
    Real
    Double
    AnyArray
      array(char)[]
    Nil
    String
    DynString
```

Figura D.1: The Green class hierarchy

```
Any
  AnyClass
    BasicStream
      InputStream
      OutputStream
      Stream
    CatchFileException
    Iter( T : Any )
    IterFilter( T : Any )
    Filter( T : Any )
    Command(T)
    Function(T)
    Container( T : Any )
      List( T : Any )
      DList( T : Any )
      Stack( T : Any )
      DStack( T : Any )
      Queue( T : Any )
      DQueue( T : Any )
      DoubleQueue( T : Any )
      DDoubleQueue( T : Any )
      Vector( T : Any )
    HashFunction( T : Any )
    Compose( T, U : Any )
    Dict( T, U : Any )
    DictIter( T, U : Any )
    IntegerSet
```

Figura D.2: Container and stream class hierarchy

AnyValue
toString() : String
getInfo() : AnyObjectInfo
getClassInfo() : ClassInfo
getClassObject() : AnyClassObject

Any
toString() : String
isObjectOf(aClass : AnyClassObject) : boolean
shallowClone() : Any (exception : CatchOutOfMemoryException)
deepClone() : Any (exception : CatchOutOfMemoryException)
shallowCopy(other : Any) : boolean
shallowEqual(other : Any) : boolean
deepEqual(other : Any) : boolean
getInfo() : AnyObjectInfo
equals(other : Any) : boolean

AnyClassObject subclassOf Any
toString() : String
isObjectOf(aClass : AnyClassObject) : boolean
shallowClone() : Any (exception : CatchOutOfMemoryException)
deepClone() : Any (exception : CatchOutOfMemoryException)
shallowCopy(other : Any) : boolean
shallowEqual(other : Any) : boolean
deepEqual(other : Any) : boolean
equals(other : Any) : boolean
getInfo() : AnyObjectInfo
getAssociateClassInfo() : ClassInfo
getInitMethod() : ObjectMethodInfo

AnyClass subclassOf Any
getClassInfo() : ClassInfo
getClassObject() : AnyClassObject

object In	
readCh()	: char
readByte()	: byte
readInteger()	: integer
readLong()	: long
readReal()	: real
readDouble()	: double
readString()	: String
readLine()	: String

object Out
write(v : ... array(Any)[])
writeln(v : ... array(Any)[])

object OutError
write(v : ... array(Any)[])
writeln(v : ... array(Any)[])

object Storage
removeFile(fileName : String) : boolean
renameFile(oldName, newName : String) : boolean
openFile(fileName : String) : integer
closeFile(fd : integer)
read(fd, n : integer; in : array(byte)[])
write(fd, n : integer; out : array(byte)[])
getError() : integer

object Runtime
exit(errorCode : integer)
putAtEndList(f : Function)
getClasses() : DS.Iter(ClassInfo)
searchForClass(name : String) : ClassInfo
getMethodCallStack() : DS.Stack(MethodCallInfo) (exception : CatchNoReflectiveCallInfoException)
getCatchObjectStack() : DS.Stack(Catch)

object Memory
sizeLargestBlock() : long
sizeFreeMemory() : long
doGarbageCollection()
collectionOn()
collectionOff()

object char
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass) (exception : CatchTypeErrorException) : char
cast(value : byte) : char
cast(value : integer) : char
castOk(any : AnyClass) : boolean
castOk(value : byte) : boolean
castOk(value : integer) : boolean
getMinValue() : char
getMaxValue() : char
getMaxIntegerChar() : integer
getMinIntegerChar() : integer

object boolean
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass) (exception : CatchTypeErrorException) : boolean
castOk(any : AnyClass) : boolean
cast(value : integer) : boolean
cast(value : byte) : boolean
getMinValue() : boolean
getMaxValue() : boolean

object byte
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass) (exception : CatchTypeErrorException) : byte
cast(value : boolean) : byte
cast(value : char) : byte
cast(value : integer) : byte
cast(value : long) : byte
cast(value : real) : byte
cast(value : double) : byte
castOk(any : AnyClass) : boolean
castOk(value : integer) : boolean
castOk(value : long) : boolean
castOk(value : real) : boolean
castOk(value : double) : boolean
getMinValue() : byte
getMaxValue() : byte

object integer
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass) (exception : CatchTypeErrorException) : integer
cast(value : char) : integer
cast(value : boolean) : integer
cast(value : byte) : integer
cast(value : long) : integer
cast(value : real) : integer
cast(value : double) : integer
castOk(any : AnyClass) : integer
castOk(value : double) : boolean
castOk(value : real) : boolean
castOk(value : long) : boolean
getMinValue() : integer
getMaxValue() : integer

object long
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass)   (exception : CatchTypeErrorException) : long
cast(value : byte) : long
cast(value : integer) : long
cast(value : real) : long
cast(value : double) : long
castOk(any : AnyClass) : boolean
castOk(value : double) : boolean
castOk(value : real) : boolean
getMinValue() : long
getMaxValue() : long

object real
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass)
(exception : CatchTypeErrorException) : real
cast(value : byte) : real
cast(value : integer) : real
cast(value : long) : real
cast(value : double) : real
castOk(any : AnyClass) : boolean
castOk(value : double) : boolean
getRadix() : integer
getRounds() : integer
getPrecision() : integer
getEpsilon() : real
getMantDig() : integer
getMinValue() : real
getMaxValue() : real
getMaxExp() : real
getMinExp() : real

object double
getSizeInBits() : integer
getSize() : integer
cast(any : AnyClass)
(exception : CatchTypeErrorException) : double
cast(value : byte) : double
cast(value : integer) : double
cast(value : long) : double
cast(value : real) : double
castOk(any : AnyClass) : double
getRadix() : integer
getRounds() : integer
getPrecision() : integer
getEpsilon() : double
getMantDig() : integer
getMinValue() : double
getMaxValue() : double
getMaxExp() : double
getMinExp() : double

AnyClass
getClassInfo() : ClassInfo
getClassObject() : AnyClassObject

Char
init(value : char)
get() : char

Boolean
init(value : boolean)
get() : boolean

Byte
init(value : byte)
get() : byte

Integer
init(value : integer)
get() : integer

Long
init(value : long)
get() : long

Real
init(value : real)
get() : real

Double
init(value : double)
get() : double

AnyArray
set(v : Any; i : integer) (exception : CatchAnyArrayException)
set(v : Any; i, j : integer) (exception : CatchAnyArrayException)
set(v : Any; i, j, k : integer; others : ... array(integer)[]) (exception : CatchAnyArrayException)
get(i : integer) : Any (exception : CatchAnyArrayException)
get(i, j : integer) : Any (exception : CatchAnyArrayException)
get(i, j, k : integer, others : ... array(integer)[]) (exception : CatchAnyArrayException)
getSize() : integer
toString() : String

array(char)[]
getSize() : integer
init(first, second, third, ... : integer)
getIter() : DS.Iter(T)
forEach(f : Function(T))
replaceBy(cmd : Command(T))
collect(f : Filter(T)) : array(T)[]
remove(f : Filter(T))
reset(up : boolean)
more() : boolean
next() : T

Nil
no methods defined in this class

String
get(i : integer) : char
getIter() : DS.Iter(char)
cmp(other : String) : integer
cmpIgnoreCase(other : String) : integer
newConcat(other : String) (exception : CatchOutOfMemoryException) : String
tocharArray(copyto : array(char)[]) (exception : CatchOutOfMemoryException)
tocharArray(copyto : array(char)[]; i : integer) (exception : CatchOutOfMemoryException)
tobyteArray(copyto : array(byte)[]) (exception : CatchOutOfMemoryException)
tobyteArray(copyto : array(byte)[]; i : integer) (exception : CatchOutOfMemoryException)
getSize() : integer
newToLowerCase() (exception : CatchOutOfMemoryException) : String
newToUpperCase() (exception : CatchOutOfMemoryException) : String
getSubset(from, to2 : integer) (exception : CatchOutOfMemoryException) : String
search(s : String) : integer
init(s : String)
hashCode() : integer
tobyte() : byte
tointeger() : integer
tolong() : long
toreal() : real
todouble() : double
toDynString() (exception : CatchOutOfMemoryException) : DynString

DynString
init(s : String)
init(s : DynString)
get(i : integer) : char
getIter() : DS.Iter(char)
cmp(other : DynString) : integer
cmpIgnoreCase(other : DynString) : integer
concat(other : String) (exception : CatchOutOfMemoryException)
tocharArray(copyto : array(char)[]) (exception : CatchOutOfMemoryException)
tocharArray(copyto : array(char)[]; i : integer) (exception : CatchOutOfMemoryException)
tobyteArray(copyto : array(byte)[]) (exception : CatchOutOfMemoryException)
tobyteArray(copyto : array(byte)[]; i : integer) (exception : CatchOutOfMemoryException)
getSize() : integer
toLowerCase()
toUpperCase()
getSubset(from, to2 : integer)
(exception : CatchOutOfMemoryException) : DynString
search(s : DynString) : integer
hashCode() : integer
tobyte() : byte
tointeger() : integer
tolong() : long
toreal() : real
todouble() : double
removeSpaceBegin()
removeSpaceEnd()
toString() (exception : CatchOutOfMemoryException) : String
prepend(toadd : DynString) (exception : CatchOutOfMemoryException)
removeAllCh(ch : char) : boolean
remove(i : integer)
insert(i : integer; ch : char) (exception : CatchOutOfMemoryException)
add(i : integer; ch : char) (exception : CatchOutOfMemoryException)
add(ch : char) (exception : CatchOutOfMemoryException)

object Meta
attachShell(any : IdentAST; exp : ExprAST)
removeShell(any : IdentAST)
attachExtension(aClass : ClassNo; dynExt : ExtensionClassNo)
removeExtension(aClass : ClassNo)

BasicStream
open(name : String; mode : integer) (exception : CatchFileException)
close() (exception : CatchFileException)
getSize() : integer (exception : CatchFileException)

InputStream subclassOf BasicStream
init(name : String) (exception : CatchFileException)
read(v : array(char)[]; n : long) (exception : CatchFileException)
read(v : array(byte)[]; n : long) (exception : CatchFileException)
read(s : DynString) (exception : CatchFileException)
readln(s : DynString) (exception : CatchFileException)

OutputStream subclassOf BasicStream
init(name : String) (exception : CatchFileException)
write(v : array(char)[]) (exception : CatchFileException)
write(v : array(char)[]; n : long) (exception : CatchFileException)
write(v : array(byte)[]) (exception : CatchFileException)
write(v : array(byte)[]; n : long) (exception : CatchFileException)
write(s : DynString) (exception : CatchFileException)
writeln(s : DynString) (exception : CatchFileException)

Stream subclassOf BasicStream
open(name : String; mode : integer)   (exception : CatchFileException)
This class has all methods of `InputStream` and `OutputStream`.

CatchFileException
init()
throw(exc : OpenFileException)
throw(exc : CloseFileException)
throw(exc : ReadFileException)
throw(exc : WriteFileException)

Iter(T : Any)
more() : boolean;
next() : T
reset()
toArray() : array(T)[]

IterFilter(T : Any)
init(piter : Iter(T); pf : Filter(T))
more() : boolean
next() : T
toArray() : array(T)[]
reset()

Filter(T : Any)
test(x : T) : boolean

Command(T)
doIt(x : T) : T

Function(T)
exec(x : T)

Container(T : Any)
init()
add(elem : T) (exception : CatchOutOfMemoryException)
get() : T (exception : CatchNotFoundException)
remove() : T (exception : CatchNotFoundException)
removeAll()
getSize() : integer
empty() : boolean
full() : boolean
getIter() : DS.Iter(T)
forEach(f : Function(T))
replaceBy(cmd : Command(T))
collect(f : Filter(T)) : Container(T)
remove(f : Filter(T))
reset()
next() : T
endIter()

List(T : Any) subclassOf Container(T)
no methods defined in this class

DList(T : Any) subclassOf Container(T)
no methods defined in this class

Stack(T : Any) subclassOf Container(T)
no methods defined in this class

DStack(T : Any) subclassOf Container(T)
no methods defined in this class

Queue(T : Any) subclassOf Container(T)
no methods defined in this class

DQueue(T : Any) subclassOf Container(T)
no methods defined in this class

DoubleQueue(T : Any) subclassOf Container(T)
addFront(elem : T) (exception : CatchOutOfMemoryException)
addBack(elem : T) (exception : CatchOutOfMemoryException)
getFront() : T (exception : CatchNotFoundException)
getBack() : T (exception : CatchNotFoundException)
removeFront() : T (exception : CatchNotFoundException)
removeBack() : T (exception : CatchNotFoundException)

DDoubleQueue(T : Any) subclassOf Container(T)
no methods defined in this class

Vector(T : Any) subclassOf Container(T)
init(p_size : integer)
init()
add(elem : T) (exception : CatchOutOfMemoryException)
add(elem : T; i : integer) (exception : CatchOutOfMemoryException)
get() : T (exception : CatchNotFoundException)
get(i : integer) : T (exception : CatchNotFoundException)
remove() : T (exception : CatchNotFoundException)
remove(i : integer) : T (exception : CatchNotFoundException)
getNum() : integer
setNum(p_num : integer)

HashFunction(T : Any)
setSize(psize : integer)
hash(elem : T) : integer

Compose(T, U : Any)
init(pt : T; pu : U)
getT() : T
getU() : U

Dict(KeyType, ValueType : Any)
init()
init(pSize : integer)
init(pSize : integer; phashFunction : HashFunction(KeyType))
add(key : KeyType; value : ValueType) (exception : CatchOutOfMemoryException)
get(key : KeyType) : U (exception : CatchNotFoundException)
remove(key : KeyType) : U (exception : CatchNotFoundException)
getSize() : integer
removeAll()
getIter() : DS.DictIter(KeyType, ValueType)

DictIter(KeyType, ValueType : Any)
init(pdict : Dict(KeyType, ValueType))
more() : boolean
next() : Container(KeyType, ValueType)

IntegerSet
init()
add(n : integer) (exception : CatchOutOfMemoryException)
remove(n : integer) (exception : CatchNotFoundException)
inSet(n : integer) : boolean
empty() : boolean
getIter() : DS.Iter(integer)
removeAll()

ClassInfo
getName() : String
toString() : String
isSupertypeOf(aSubtype : ClassInfo) : boolean
isSuperclassOf(aSubclass : ClassInfo) : boolean
getSuperclass() : ClassInfo
getInstanceVariables() : DS.Iter(ClassInstanceVariableInfo)
getMethods() : DS.Iter(ClassMethodInfo)
getPublicMethods() : DS.Iter(ClassMethodInfo)
getInstanceVariable(name : String) : ClassInstanceVariableInfo
getMethod(name : String; paramTypes : array(ClassInfo)[]) : ClassMethodInfo
getMethod_v(name : String; paramTypes : ... array(AnyClassObject)[]) : ClassMethodInfo
getPublicMethod(name : String) : DS.Iter(ClassMethodInfo)
getThisClassInstanceVariables() : DS.Iter(ClassInstanceVariableInfo)
getThisClassMethods() : DS.Iter(ClassMethodInfo)
getThisClassPublicMethods() : DS.Iter(ClassMethodInfo)
getAssociateClassObject() : AnyClassObject
isClassOf(any : Any) : boolean
isAbstract() : boolean
getNotes() : DS.Iter(CodeAnnotations)

ValueClassInfo subclassOf ClassInfo
no methods defined in this class

RefClassInfo subclassOf ClassInfo
no methods defined in this class

AbstractClassInfo subclassOf RefClassInfo
no methods defined in this class

ConcreteClassInfo subclassOf RefClassInfo
no methods defined in this class

NormalClassInfo subclassOf ConcreteClassInfo
no methods defined in this class

ArrayClassInfo subclassOf ConcreteClassInfo
getArrayElementClass() : ClassInfo
getNumberOfDimensions() : integer
toString() : String

MethodInfo
getName() : String
toString() : String
getBodyInfo() : MethodBodyInfo (exception : CatchNoReflectiveBodyInfoException)
getVisibility() : integer
getParameterTypes() : DS.Iter(ClassInfo)
getReturnType() : ClassInfo
getParameters() : DS.Iter(ParameterInfo)
getExceptionClass() : ClassInfo
getNotes() : DS.Iter(CodeAnnotations)

ClassMethodInfo subclassOf MethodInfo
isAbstract() : boolean
invoke(obj : Any; args : array(Any)[]) : Any (exception : CatchInvokePackedException)
invoke_v(obj : Any; args : ... array(Any)[]) : Any (exception : CatchInvokePackedException)

ObjectMethodInfo subclassOf MethodInfo
invoke(args : array(Any)[]) : Any (exception : CatchInvokePackedException)
invoke_v(args : ...array(Any)[]) : Any (exception : CatchInvokePackedException)

InstanceVariableInfo
getName() : String
getType() : ClassInfo
getNotes() : DS.Iter(CodeAnnotations)

ClassInstanceVariableInfo subclassOf InstanceVariableInfo
set(obj : Any; v : Any) (exception : CatchTypeErrorException)
get(obj : Any) : Any (exception : CatchTypeErrorException)

ObjectInstanceVariableInfo subclassOf InstanceVariableInfo
set(v : Any) (exception : CatchTypeErrorException)
get() : Any

AnyObjectInfo
getObject() : Any
getInstanceVariables() : DS.Iter(ObjectInstanceVariableInfo)[]
getMethods() : DS.Iter(ObjectMethodInfo)
getPublicMethods() : DS.Iter(ObjectMethodInfo)
getPublicMethod(name : String) : DS.Iter(ObjectMethodInfo)
getMethod(name : String; paramTypes : array(ClassInfo)[]) : ObjectMethodInfo
getMethod(name : String) : ObjectMethodInfo
getInstanceVariable(name : String) : ObjectInstanceVariableInfo
getTypeInfo() : ClassInfo

ObjectInfo subclassOf AnyObjectInfo
no methods defined in this class

ClassObjectInfo subclassOf AnyObjectInfo
new(args : array(Any)[]) (exception : CatchNewException) : Any
new_v(args : ... array(Any)[]) (exception : CatchNewException) : Any
getNotes() : DS.Iter(CodeAnnotations)

MethodBodyInfo
getMethodInfo() : MethodInfo
getAssertionInfo() : AssertionInfo
getLocalVariables() : DS.Iter(VariableInfo)
getStatements() : DS.Iter(StatementNo)

AssertionInfo
getBeforeExpression() : ExprAST
getAfterExpression() : ExprAST
getStatVarDeclarations() : DS.Iter(StatVarDecNo)

VariableInfo
getName() : String
getType() : ClassInfo
getDeclaringMethod() : MethodInfo
getNotes() : DS.Iter(CodeAnnotations)

ParameterInfo subclassOf VariableInfo
isVariableNumber() : boolean

MethodCallInfo
getMethodInfo() : MethodInfo
getLiveLocalVariables() : DS.Iter(LiveLocalVariableInfo)
getLiveParameters() : DS.Iter(LiveParameterInfo)

LiveLocalVariableInfo
getVariableInfo() : VariableInfo
set(v : Any) (exception : CatchTypeErrorException)
get() : Any

LiveParameterInfo subclassOf LiveLocalVariableInfo
no methods defined in this class

Referências Bibliográficas

[1] America, Pierre and Linden, Frank van Der. A Parallel Object-Oriented Language with Inheritance and Subtyping. *SIGPLAN Notices*, Vol. 25, No. 10, pp. 161–168, October 1990. ECOOP/OOPSLA 1990.

[2] Cardelli, Luca. A Semantics of Multiple Inheritance. Colloquium on the Semantics of Data Types, In LNCS # 173, pp. 51–67, Springer-Verlag.

[3] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[4] Guimarães, José de Oliveira. Reflection for Statically Typed Languages. *XII European Conference on Object-Oriented Programming*. Published as LNCS No. 1445, 1998.

[5] Guimarães, José de Oliveira. Suggestion for a Parametrized Class Model. *OOPS Messenger*, Vol. 5, No. 3, pp. July 1994.

[6] Henderson, Robert and Zorn, Benjamin. A Comparison of Object-Oriented Programming in Four Modern Languages. *Software-Practice and Experience*, Vol. 24 No.11, pp. 1077-1095, November 1994.

[7] Horn, Chris. Conformance, Genericity, Inheritance and Enhancement. Proceedings of ECOOP 87, pp 223–233, Published as LNCS 272.

[8] Java and Internet Programming. Dr. Dobb's Journal, Vol. 20, No. 8, pp. 56-60, August 1995.

[9] Liskov, Barbara and Snyder, Alan and R. Atkinson and C. Schaffert. Abstraction Mechanisms in CLU. CACM, Vol. 20, No. 8, pp. Aug. 1977.

[10] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[11] Meyer, Bertrand. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Notices*, Vol. 22, No. 2, pp. 85-94, February 1987.

[12] Raj, R. K. and Tempero, Ewan and Levy, Henry and Black, A. P. Emerald: A General Purpose Programming Language. *Software Practice and Experience*, Vol. 21, No. 1, pp 91-118, January 1991.

[13] Rodrigues, Noemi and Rangel, José and Ierusalimschy, Roberto. Types in School. ACM SIGPLAN Notices, Vol. 28, No. 8, pp 81-89, August 1993.

[14] Snyder, Alan. Encapsulation and Inheritance in Object-Oriented Programming Languages. *SIGPLAN Notices*, Vol. 21, No. 11, pp. 38-45, November 1986. OOPSLA 86.

[15] Snyder, Alan. Inheritance and the Development of Encapsulated Software Components. In *Research Directions in Object-Oriented Programming*, pp. 165-188. MIT Press, 1987.

[16] Stephen Omohundro. The Sather Programming Language. Dr. Dobb's Journal, Vol. 18, No. 11, pp. 42–49, October 1993.

[17] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.