Chapter 13

Implementing the Object-Oriented Paradigm

In this chapter we will discuss some of the difficulties involved in implementing those features of Leda that are relevant to the object-oriented paradigm.

13.1 Memory Layout

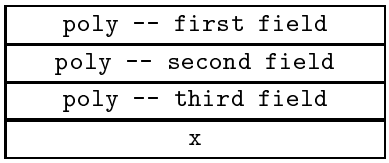
In Chapter 10 we introduced the concept of a polymorphic variable. That is, in an object-oriented language it is possible to declare a variable which can hold a number of different types of values over the course of execution. The only limiting restriction in this regard is that all such values must be instances of classes which inherit from a single common ancestor class, which must be the class used in the declaration of the polymorphic variable. In Chapter 12 we discussed some of the many uses for such values.

The presence of polymorphic variables introduces a number of interesting problems for the language implementor, problems which are not found in the implementation of more conventional languages. The first such difficulty we will consider is concerned with the allocation of memory space for variables and for values.

In a conventional language, variables are allocated a fixed amount of space in a fixed location in memory, or in a fixed location in a block of memory called an *activation record* that is allocated once at the beginning of a function invocation, and released when a function terminates.

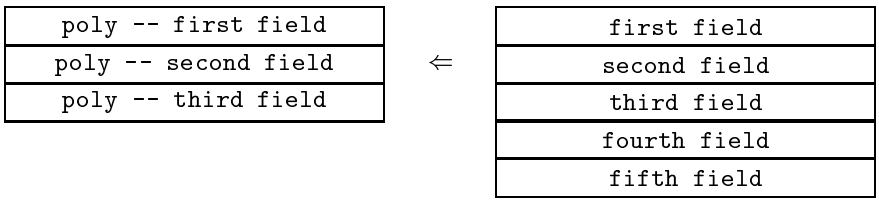
Suppose, for example, that we imagine a function which uses one local integer variable named `x`, and also declares one local variable named `poly` which is a record containing three integer data fields. An activation record for such a procedure might look as follows:[1]

[1] In practice activation records also contain space for parameter values, and for various "bookkeeping"

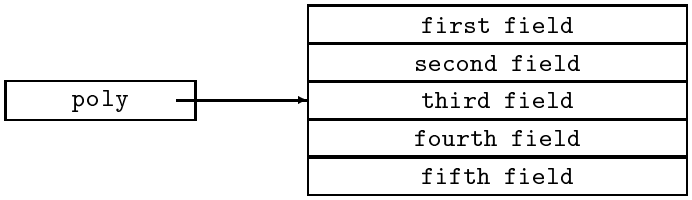


Now imagine that `poly` is not a simple record structure, but is instead a polymorphic variable declared as being an instance of a class which defines three integer data fields. The defining characteristic of polymorphic variables was that they can hold *values* that were generated from subclasses. So next imagine that a subclass of the class from which `poly` was declared defines two additional integer data fields, and that an attempt is made to assign a value generated by this subclass to the variable `poly`.

The value on the right of the assignment contains five integer data fields. The memory allocated to `poly` contains space only for three integer data fields. Simply put, the problem is that we are trying to store more information into a fixed size box than it can hold:



The solution in Leda is to eliminate altogether the idea that a variable defines a fixed size box.[2] Instead, all variables are in reality pointers. When a value is assigned to a variable, only this pointer field is changed. The value to which it points can then be any size whatsoever, with no limitations being imposed at compile time.



Unfortunately, a negative consequence of this decision is that it naturally leads to the pointer semantics for assignment which, as we noted at the beginning of Chapter 10, can occasionally be somewhat confusing.

values, such as return address fields. These details are unimportant for our discussion here.

[2] Note that this is not the only solution. The language C++, for example, takes an entirely different approach, simply slicing off the extra fields during assignment.

13.2 Dynamic Allocation

It is common that a value can be created and bound to a variable in a way that ensures the value will outlive the context in which it is created. This can occur, for example, if a value is assigned to a variable from a surrounding context. To illustrate, consider the following program, which uses the functional version of the list abstraction from Chapter 4:

```
var
    aList : List[integer];

    function escape ();
    begin
        aList := List[integer](17, emptyList);
    end;
```

The value created inside the function `escape` must exist even after the function has returned. It is for this reason that very few values created in Leda can be allocated in a stack-like fashion, as is common in languages such as Pascal or C. Instead, all values in Leda are dynamically allocated on a heap, and must be reclaimed by a memory management mechanism, such as a garbage collection system.

13.3 Class Pointers

An important property of polymorphic variables is that the selection of which of many possible versions of an overridden function to invoke in any particular situation depends upon the actual, run-time or *dynamic* type held by such a variable, and not on the defined, or *static* type with which it was declared. Thus, it is a requirement of all object-oriented languages that all such values possess at least a rudimentary form of "self-knowledge" concerning their own type, and be able to use this knowledge in the selection of a function to execute.

In Leda this self-knowledge is embodied in a data field that declared in class `object`, and thus is common to all objects (see Figure 13.1). This field is declared as an instance of class `Class`. Class `Class` maintains information specific to each defined class. In particular, this information includes the name of the class as a string, the number of data fields defined in each class (that is, the size of each instance), and a pointer to the parent class.

The method `isInstance` takes as argument a value, and returns true if the value is an instance of a given class (either directly or through inheritance). To do this, the function makes use of the relational operators, which have been redefined to indicate the class-subclass relationship. The less-than operator takes two classes, and returns true if the parent class of the left argument, or any ancestor of this parent, is the same as the right argument. The default meaning of the equality operator indicates whether the two arguments (that is,

```
class object;
var
    classPtr : Class;     { pointer describing object }

    ...
end;

class Class of ordered[Class];
var
    name : string;
    size : integer;
    parent : Class;

    function asString()->string;
    begin
        return name;
    end;

    function less (arg : Class)->boolean;
    begin
        if self == arg then    { equal, not less }
            return false;
        if parent == arg then
            return true;
        return parent <> self & parent < arg;
    end;

    function isInstance (val : object)->boolean;
    begin
        return val.classPtr <= self;
    end;
end;

function typeTest [T : object] (val : object, aClass : Class)->T;
begin
    if aClass.isInstance(val) then
        return cfunction Leda_object_cast(val)->T;
    return NIL;
end;
```

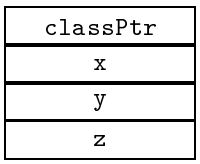
Figure 13.1: The class pointer and the class `Class`

two classes) are identically the same. As we saw in Chapter 12, the remaining relational operators are all defined in terms of these two functions.

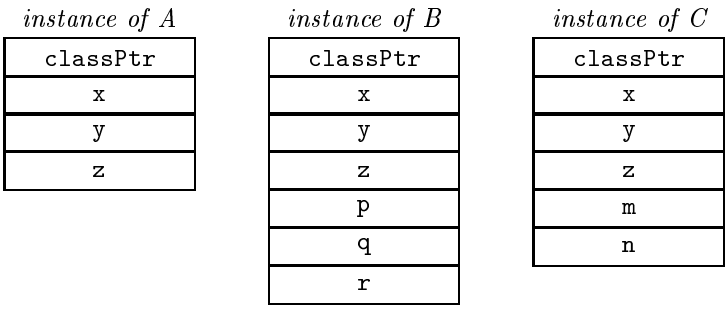
The function `isInstance` uses the ability to do relational tests on classes in order to determine if an argument value is an instance of the receiver class. The `isInstance` function is utilized in one of the more unusual functions provided as part of the standard run-time library. This function is named `typeTest`, and is used to perform "reverse polymorphism"; that is, to take a value declared as holding an instance of a parent class, and determine whether or not it is, in fact, maintaining a value generated from a given child class. The `cfunction` invoked to perform the type conversion in this situation performs no action, and merely serves to foil the type checking mechanism. If the argument value is not an instance of the given class, then the undefined value `NIL` is returned.

13.4 Subclasses as Extensions

Data fields defined within a class in Leda are handled in a similar fashion to data fields in records in languages such as Pascal or C. That is, the data fields are simply catenated together in a contiguous fashion in memory, end to end, to yield a block of values which together constitute the state of the object. For example, suppose an object represents an instance of a class `A` in which are defined three data values, `x`, `y` and `z`. We can imagine `A` looking something like the following:



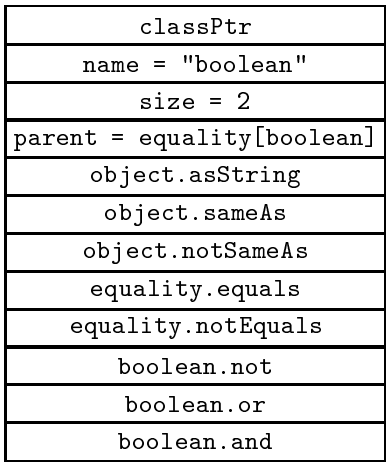
An important feature of subclassing is that a child class strictly extends the number of data fields held by each instance, never decreases this size. Furthermore, and just as importantly, we can arrange so that the location of each field inherited from a parent class is found in the same location relative to the start of the object, regardless of the class from which the instance was generated. That is, suppose we now imagine a subclass of `A` named `B` which defines three new data fields `p`, `q` and `r`, and a second subclass of `A` named `C` which defines two fields `m` and `n`. A value from each of these classes can be imagined as looking something like the following:



It is because the location of the inherited data fields are known to be fixed, regardless of the class from which the items were generated, that the compiler is able to produce code for functions defined in the parent classes. That is, class `A` can define functions which manipulate the three values `x`, `y` and `z`. These functions will continue to operate even if they are manipulating an instance of `B` or `C`, instead of an instance of class `A`.

13.5 Virtual Dispatch Tables

A feature similar to the extension of subclass data areas from the data areas generated by parent classes is involved in the technique used to implement the mechanism of function overriding. The actual representation of an instance of class `Class` includes more than simply the data fields defined in that class, but is extended to include, as well, fields containing pointers to functions which are implemented in the given class. That is, the instance of class `Class` containing information about the class `boolean` in the standard library looks something like the following:



The data fields `classPtr`, `name`, `size`, and `parent` are generated by the variable declarations in class `object` and `boolean`. The remaining fields do not contain data values, but pointers to functions. The first three are pointers to functions implemented in class `object`, the next two are derived from functions defined in class `equality`, while the last three are associated with functions in class `boolean`. This table of functions is traditionally known as a *virtual dispatch table* (This is because in other object-oriented languages overridden functions are described as "virtual", and the table is used to dispatch execution on such functions.)

Subclasses can inherit functions, they can override existing functions, and they can implement new functions. The class `True`, for example, inherits a number of functions from class `boolean`, and overrides four. The virtual dispatch table generated for this class would look as follows:

classPtr
name = "True"
size = 2
parent = boolean
True.asString
object.sameAs
object.notSameAs
equality.equals
equality.notEquals
True.not
True.or
True.and

There are several features to note. The first is that inherited and overridden functions are found in the same location in both the table generated for class `boolean` and that generated for class `True`. In exactly the same way that the common location of data areas permits a compiler to generate code that will work for any data value derived from a given class, the same feature in the class table means that to invoke an inherited function it is simply sufficient to execute the function found at a fixed location in the class table. Furthermore, to override a function it is sufficient to simply replace the pointer in the class table with the pointer to the new function. All functions that are not so replaced are then inherited without change.

An invocation of a function inherited in a class structure is converted, using this mechanism, into an invocation of the function found by indexing the virtual dispatch table by a fixed amount that can be determined at compile time. For example, to produce the printable representation of a value the dispatch table is indexed into location 4 (index values start at zero). The function found there will be the operation appropriate for the value being manipulated, whether the value is a boolean, an integer, a string, or any other type. Note that in each of these cases a different function will be invoked, as each class overrides the method `asString` in a different manner.

Notes and Bibliography

I discuss a variety of different implementation techniques for object-oriented languages in more detail in my earlier book on object-oriented programming [Budd 91b]. Other explanations can be found in [Cox 86, Ellis 90].

Many object-oriented languages include a feature called "multiple inheritance" with which a class can inherit features from two or more parent classes. There are three reasons

why I have not included this feature in Leda. The first is that the semantics of multiple inheritance are not at all clear in all situations (see [Budd 91b] or [Sakkinen 92] for a fuller discussion of this point). The second is that in practice programs that use multiple inheritance can usually be replaced by programs that use single inheritance that are just as concise and clear, if not more so, than their multiple counterparts. Finally, the implementation of multiple inheritance is considerably more difficult than the implementation of single inheritance. In single inheritance, the initial portion of a class always has the same structure as the structure of its parent from which it inherits features. But if a class inherits from two or more classes, its initial portion can match the structure of one or the other parent, but cannot match both.

Chapter 18

Implementing the Functional Paradigm

In this chapter we explore some of the problems that will be encountered by the language implementor in considering those aspects of the language related to functional programming.

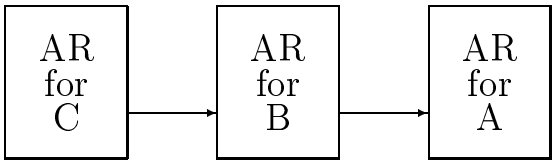
18.1 Activation Records

As we noted already, in the earlier chapter which dealt with the implementation issues in object-oriented languages, every time a function or procedure is started, a block of memory called an *activation record* is allocated. This activation record contains:

- Space for local variables. To be precise, sufficient space to hold a pointer for each local variable, since we saw in Chapter 13 that each variable is maintained internally simply by a pointer.
- Space for pointers to each parameter value.
- Space for "bookkeeping" information. Although the details of this last category will differ in each implementation, this information might consist of, for example, a return address, a pointer to the calling activation record, saved values of registers, and so on.

While the exact address of a variable in memory can not be determined by a compiler, the offset of the variable pointer relative to the start of an activation record can be computed. By this means the code generated by the compiler can, at run time, efficiently gain access to local variable values, simply by maintain a single pointer to the current activation record. Note that this remains true even in the case of recursive procedures.

As part of the process of function invocation, every activation record is linked to the activation record for the procedure which performed the function call. This connection is called the *dynamic link*. Imagine that a procedure `A` calls a procedure `B`, which in turn calls a procedure `C`. The activation record structure will, at the point where we execute procedure `C`, look like the following:



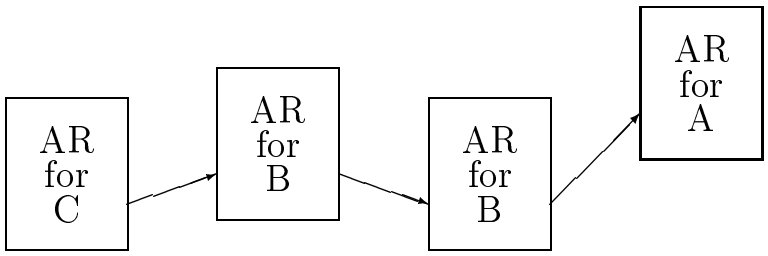
18.2 Nested Procedures

We noted that by maintaining a pointer to the current activation record the values of local variables can quickly and easily be accessed. Now consider the problem of accessing variables from surrounding scopes, for example from functions in which other functions have been nested:

```
function A();
var  x : integer;

    function B();
    var y : integer;

        function C();
        var z : integer;
        begin
            z := x + y;
        end;

    begin
        ...
        if x < y then
            B()
        else
            C();
    end;
```

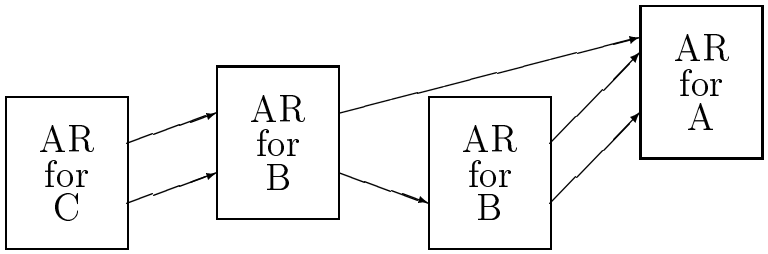
```
begin
    B();
end;
```

Imagine we are executing the assignment statement in the procedure `C`. The assignment statement uses a local variable, as well as a variable found in the activation record for `B` and one in the activation record for `A`. One could imagine that it would be possible to use the dynamic link to obtain the values for these latter variables, but this is more difficult than it might at first seem. For example, what happens if the procedure `B` is recursive? When executing `C`, the sequence of dynamic links would look something like the following:

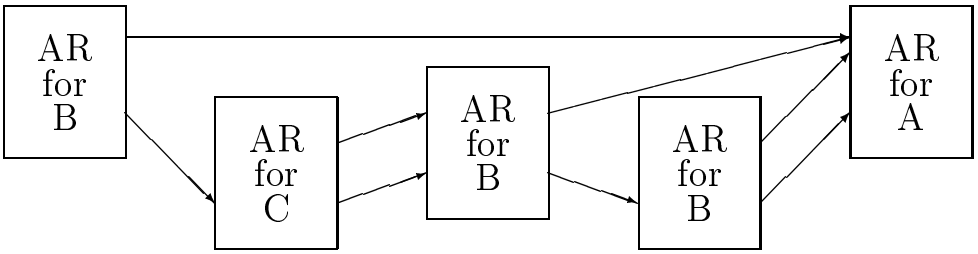


There will be an indefinite number of activation records for `B` between the activation record for `C` and the record for `A`. Hence the compiler will not be able to know how many links to traverse backwards in order to reach the activation record in which the variable `x` is being maintained.

The standard solution in this case is for activation records to maintain an entirely separate link to be used for data access. This link is called the *static link* (because it is based on the static layout of the program, not the dynamic run-time execution.) The setting of a static link is the responsibility of the *calling* procedure, not the *called* procedure.



To illustrate the way in which static links are manipulated, now imagine that the procedure `C` calls the surrounding procedure `B`. The sequence of activation records would then look as follows:



The invariant being preserved is that the static links must always point to the action record associated with the nearest surrounding context. With this information it is possible to determine, at compile time, the number of static links necessary to traverse in order to locate any variable.

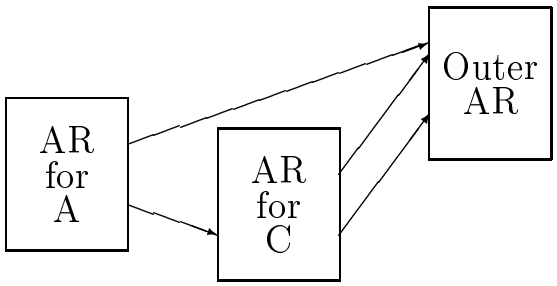
18.3 Functions as Arguments

When functions are used as arguments they introduce a new twist into the idea of data access and static links. Consider an example of such a function:

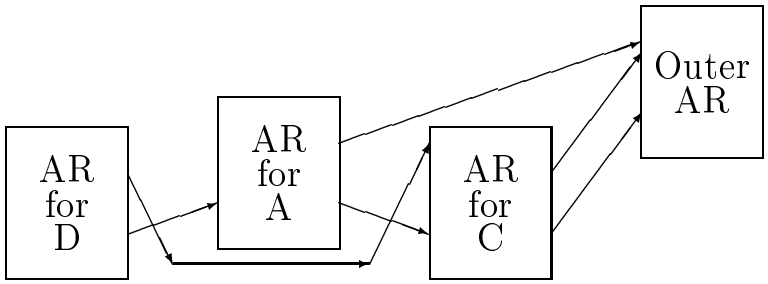
```
function A (B : function());
begin
    B();
end;

function C();
var  x : integer;

    function D();
    begin
        print(x);
    end;
begin
    x := 7;
    A(D);
end;
```

Here `A` is a function which takes as argument another function, which is locally known by the name `B`. The function `C` calls `A`, and passes to `A` a nested function, named `D`.

Here is a snapshot of the activation record layout at the point inside `A`, where `B` (that is, the procedure `D`) is about to be called:



Here is the same picture immediately after the procedure `B` (that is, `D`) is invoked:



The difficulty is that the procedure `D` must establish its static link, that is, the pointer to its surrounding context. But `A`, the procedure which calls `B` (`D`), does not know the proper context. Indeed, the proper context is not even on the link of static links which start at `A`.

In order to permit the static link to be properly maintained, a function that is passed as an argument must carry with it two things:

- A pointer to the code to be executed.
- The value of the surrounding context to use as the static link when the procedure is executed.

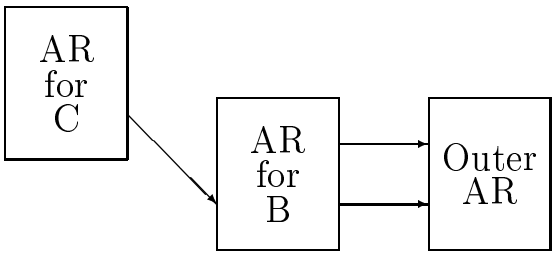
This pair is known as a *closure*. The fact that closures must be constructed for functions used as arguments is sometimes known as the *downward funarg* problem. (Funarg is short for "functional argument").

18.4 Functions used as Result

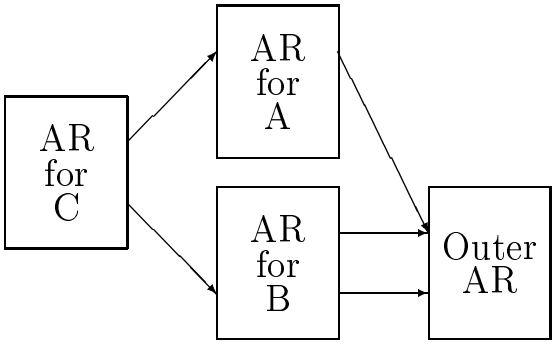
Now consider what happens when a function is returned as the result of another function, as in the following example:

```
function A ()->function();
var x : integer;
begin
    return function();
        begin
            print(x);
        end;
end;

function B();
const
    C := A();
begin
    C();
end;
```

At the point where the function `C` is invoked, the link of activation records looks something like the following:



Notice we have a problem here. The surrounding context for `C` should be the activation record for `A`. But the activation record for `A`, since `A` has returned and is no longer executing, has disappeared. Thus, functions returned as results, like functions passed as arguments, must carry with them their own context. That is, the value returned by `A` must keep the pointer to the activation record for `A` in a closure.



Worse, the fact that the variables in `A` can now be accessed even after `A` itself has ceased to be executed means that activation records cannot be allocated and eliminated in a strict stack-like fashion, as they can if function results are not permitted. Instead, activation records for such functions must be allocated on a heap, just like any other dynamic value.

The fact that function results must carry with them their context, and that such usage then demands that context values not be destroyed immediately after a function call completes, is sometimes known as the *upward funarg* problem.

18.5 Tail Recursion Elimination

Because contexts must, in general, be dynamically allocated (and eventually, recovered as garbage), it is worthwhile to search for situations where the allocation of a context is unnecessary. Fortunately, one of the more common situations is also very easy to discover.

It is often the case that a function will be terminated, at least some of the time, by a recursive call on the same procedure, differing from the current call only in the arguments. An example of this is the function `filter` we introduced in Chapter 14.

```
function filter [X : equality]
    (aList : List[X], pred : function(X)->boolean)->List[X];
begin
    if empty[X](aList) then
        return aList;
    if pred(head[X](aList)) then
        return List[X](head[X](aList), filter[X](tail[X](aList), pred))
    else
        return filter[X](tail[X](aList), pred);
end;
```

Since the value yielded by the recursive call in the final statement will be immediately returned by the currently executing function, there is no further need for the data values being held in the current context. Rather than allocating a new context for the recursive call, we can simply replace the parameter locations with their new values, and begin execution of the `filter` function once more, recycling the current context for use in the new invocation.

By means of this optimization, which is known as *tail recursion*, the overhead involved in a recursive function call becomes simply the cost of replacing the parameter values and the execution time associated with an unconditional branch. Often this can make a recursive procedure even more efficient than the equivalent functionality would be if expressed as a loop.

18.6 Implementation of By-Name Parameters

When an expression is passed through an argument using the mechanism of by-name parameter passing, the evaluation of the parameter is delayed until the argument value is actually used. For example, consider the following simple program:

```
var
    x : integer;

    function A (byName b : integer);
    begin
        x := 7;
        print(b);
    end;


begin
    x := 3;
    A (x + 1);
end;
```

The problem will print the value 8, not the value 4. This is because the value of `b` is computed at the point it is used, in this case when it is used as argument for the function `print`. But `b` is simply a name for the expression `x + 1`. At the point the `print` function is invoked the value of `x` is not the value it held at the time the procedure `A` was invoked (namely, 3), but has been altered by the assignment to the value 7. Thus the expression `x + 1` yields 8.

To implement pass by-name, the expression passed as argument is in fact turned into a function. In essence, it is as if the program above were instead written as follows:

```
var
    x : integer;

    function A (b : function()->integer);
    begin
        x := 7;
        print(b());
    end;


begin
    x := 3;
    A (function(); return x + 1; end);
end;
```

That is, the argument expression is turned into a function. The parameter is also interpreted as a function, and finally every use of the by-name variable is turned into a function call. The nameless, argument-less function created in this implementation of by-name parameter passing is traditionally known as a *thunk*.[1]

The by-name parameter passing mechanism in Leda is simplified by the fact that such values cannot be used as the target of an assignment. In languages in which by-name parameters can be assigned, such as Algol-60, the thunk must return, not the actual argument value, but a pointer to the argument variable.

Notes and Bibliography

More detailed descriptions if the implementation techniques used in conventional systems can be found in any good text on compiler construction, such as [Aho 86]. Recently, a number of functional programming languages have experimented with an entirely different approach, based on a technique called *graph rewriting*. A good explanation of the graph rewriting scheme is presented in [Peyton Jones 87]. It is generally thought that allocating contexts on a stack produces faster executions than is possible with the allocation of contexts on a heap. However Appel shows [Appel 87], that at least under certain circumstances this need not be the case.

[1] The name is said to have been inspired by the sound made by a programmers head hitting a wall.